

# Xpand Language Reference

Sven Efftinge, [sven@efftinge.de](mailto:sven@efftinge.de), [www.efftinge.de](http://www.efftinge.de)

Clemens Kadura, [zAJKa2309@email.de](mailto:zAJKa2309@email.de)



## Table of Contents

<b>INTRODUCTION.....</b>	<b>4</b>
<b>TEMPLATE FILES AND ENCODING.....</b>	<b>4</b>
<b>GENERAL STRUCTURE OF TEMPLATE FILES.....</b>	<b>4</b>
<b>STATEMENTS OF THE XPAND LANGUAGE.....</b>	<b>5</b>
IMPORT.....	5
EXTENSION.....	5
DEFINE.....	5
FILE.....	7
EXPAND.....	7
<i>Names.....</i>	8
FOR vs. FOREACH.....	8
<i>Specifying a Separator.....</i>	8
FOREACH.....	9
IF.....	9
PROTECT.....	10
LET.....	10
ERROR.....	11
COMMENTS.....	11
EXPRESSION STATEMENT.....	11
CONTROLLING GENERATION OF WHITESPACE.....	12
<b>ASPECT-ORIENTED PROGRAMMING IN XPAND.....</b>	<b>12</b>
JOIN POINT AND POINT CUT SYNTAX.....	12
<i>Definition Name.....</i>	13
<i>Parameter Types.....</i>	13
<i>Target Type.....</i>	13
PROCEEDING.....	13
<b>GENERATOR WORKFLOW COMPONENT.....</b>	<b>14</b>
MAIN CONFIGURATION.....	14
ENCODING.....	14
METAMODEL.....	15
OUTPUT CONFIGURATION.....	15
BEAUTIFIER.....	16
<i>JavaBeautifier.....</i>	16
<i>JavaBeautifier2.....</i>	17

<i>XmlBeautifier</i> .....	17
PROTECTED REGION CONFIGURATION.....	17

## Introduction

The *openArchitectureWare* framework contains a special language called *Xpand* that is used in templates to control the output generation. This documentation describes the general syntax and semantics of the *Xpand* language.

Typing the guillemets (« and ») used in the templates is supported by the Eclipse editor: which provides keyboard shortcuts with *Ctrl+<* and *Ctrl+>*.

## Template files and encoding

Templates are stored in files with the extension *xpt*. Template files must reside on the Java classpath of the generator process.

Almost all characters used in the standard syntax are part of `ASCII` and should therefore be available in any encoding. The only limitation are the tag brackets (guillemets), for which the characters « (Unicode `00AB`) and » (Unicode `00BB`) are used. So for reading templates an encoding should be used that supports these characters (e.g. `ISO-8859-1` or `UTF-8`).

Names of properties, templates, namespaces etc. must only contain letters, numbers and underscores.

## General structure of Template files

Here is a first example of a template.

```
«IMPORT meta::model»
«EXTENSION my::ExtensionFile»

«DEFINE javaClass FOR Entity»
  «FILE fileName()»
  package «javaPackage()»;

  public class «name» {
    // implementation
  }
«ENDFILE»
«ENDEDEFINE»
```

A template file consists of any number of `IMPORT` statements, followed by any number of `EXTENSION` statements, followed by one or more `DEFINE` blocks (called definitions).

---

## Statements of the Xpand language

### IMPORT

If you are tired of always typing the fully qualified names of your types and definitions, you can import a namespace using the IMPORT statement.

```
«IMPORT meta::model»
```

This one imports the namespace `meta::model`. If your template contains such a statement, you can use the unqualified names of all types and template files contained in that namespace. This is similar to a Java import statement `import meta.model.*`

### EXTENSION

Metamodels are typically described in a structural way (graphical, or hierarchical, etc.) . A shortcoming of this, is that it's difficult to specify additional behaviour (query operations, derived properties, etc.). Also, it's a good idea not to pollute the meta model, with target platform specific information (e.g. Java type names, packages, getter and setter names, etc.).

Extensions provide a flexible and convenient way of defining additional features of meta classes. You do this by using the *Extend* language. (See the corresponding reference documentation for details)

An EXTENSION import points to the *Extend* file containing the required extensions:

```
«EXTENSION my::ExtensionFile»
```

Note that extension files have to reside on the Java classpath, too. Therefore they use the same namespace mechanism (and syntax) as types and template files.

### DEFINE

The central concept of *Xpand* is the DEFINE block, also called a template. This is the smallest identifiable unit in a template file. The tag consists of a name, an optional comma separated parameter list as well as the name of the meta model class for which the template is defined.

```
«DEFINE templateName(formalParameterList) FOR MetaClass»  
  a sequence of statements  
«ENDDFINE»
```

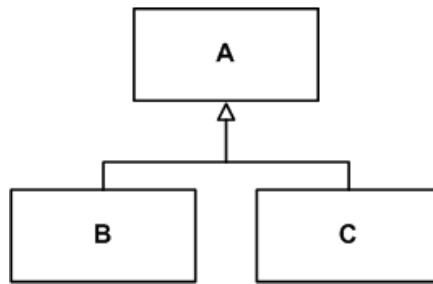
To some extent templates can be seen as special methods of the meta class – there is always an implicit *this* parameter which can be used to address the „underlying“ model element; in our example above, this model element is “MetaClass”.

As in Java a formal parameter list entry consists of the type followed by the name of that parameter.

The body of a template can contain a sequence of other statements including any text.

A full parametric polymorphism is available for templates. If there are two templates with the same name that are defined for two meta classes which inherit from the same super class, *Xpand* will use the corresponding subclass template in case the template is called for the super class. Vice versa the super class's template would be used in case a subclass template is not available. Note that this not only works for the target type, but for all parameters. Technically, the target type is handled as the first parameter.

So, assume you have the following metamodel:



Assume further, you'd have a model which contains a collection of *A*, *B* and *C* instances in the property *listOfAs*. You can then write the following template:

```

«DEFINE someOtherDefine FOR SomeMetaClass»
  «EXPAND implClass FOREACH listOfAs»
«ENDDDEFINE»

«DEFINE implClass FOR A»
  // this is the code generated for the super class A
«ENDDDEFINE»

»»«DEFINE implClass FOR B»
  // this is the code generated for the subclass B
«ENDDDEFINE»

«DEFINE implClass FOR C»
  // this is the code generated for the subclass C
«ENDDDEFINE»
  
```

So for each *B* in the list, the template defined for *B* is executed, for each *C* in the collection the template defined for *C* is invoked, and for all others (which are then instances of *A*) the default template is executed.

## FILE

The `FILE` statement redirects the output generated from its body statements to the specified target.

```
«FILE expression [outletName]»
  a sequence of statements
«ENDFILE»
```

The target is a file in the file system whose name is specified by the expression (relative to the specified target directory for that generator run). The expression for the target specification can be a concatenation (using the `+` operator). Additionally you can specify an identifier (a legal Java identifier) for the name of the outlet. (See the configuration section for a description of outlets).

The body of a `FILE` statement can contain any other statements. Example:

```
«FILE InterfaceName + ".java"»
  package «InterfacePackageName»;

  /* generated class! Do not modify! */
  public interface «InterfaceName» {
    «EXPAND Operation::InterfaceImplementation FOREACH      Operation»
  }
«ENDFILE»

«FILE ImplName + ".java" MY_OUTLET»
  package «ImplPackageName»;

  public class «ImplName» extends «ImplBaseName»
                        implements «InterfaceName» {
    //TODO: implement it
  }
«ENDFILE»
```

## EXPAND

The `EXPAND` statement „expands“ another `DEFINE` block (in a separate Variable context), inserts its output at the current location and continues with the next statement. This is similar in concept to a subroutine call.

```
«EXPAND definitionName [(parameterList)]
  [FOR expression | FOREACH expression [SEPARATOR expression] ]»
```

The various alternative syntaxes are explained below.

## Names

If the *definitionName* is a simple unqualified name, the corresponding `DEFINE` block must be in the same template file.

If the called definition is not contained in the same template file, the name of the template file must be specified. As usual, the double colon is used to delimit namespaces.

```
«EXPAND TemplateFile::definitionName FOR myModelElement»
```

Note, that you would need to import the namespace of the template file (if there is one). For instance, if the template file resides in the java package `my.templates`, there are two alternatives. You could either write

```
«IMPORT my::templates»  
...  
..«EXPAND TemplateFile::definitionName FOR myModelElement»  
..
```

OR

```
«EXPAND my::templates::TemplateFile::definitionName  
FOR myModelElement»
```

## FOR vs. FOREACH

If `FOR` or `FOREACH` is omitted the other template is called `FOR this`.

```
«EXPAND TemplateFile::definitionName»
```

equals

```
«EXPAND TemplateFile::definitionName FOR this»
```

If `FOR` is specified, the definition is executed for the result of the target expression.

```
«EXPAND myDef FOR entity»
```

If `FOREACH` is specified, the target expression must evaluate to a collection type. The other definition is executed for each element of that collection.

```
«EXPAND myDef FOREACH entity.allAttributes»
```

## Specifying a Separator

If a definition is to be expanded `FOREACH` element of the target expression it's possible to specify a `SEPARATOR` expression:

```
«EXPAND paramTypeAndName FOREACH params SEPARATOR ", "»
```

The result of the separator expression will be written to the output between each evaluation of the target definition (not *after* each one, but rather only in *between* two elements. This comes in handy for things such as comma-separated parameter lists).

An `EvaluationException` will be thrown if the specified target expression cannot be evaluated to an existing element of the instantiated model or no suitable `DEFINE` block can be found.

## FOREACH

This statement expands the body of the `FOREACH` block for each element of the target collection that results from the expression. The current element is bound to a variable with the specified name in the current context.

```
«FOREACH expression AS variableName [ITERATOR iterName] [SEPARATOR  
expression]»  
  a sequence of statements using variableName to access the  
  current element of the iteration  
«ENDFOREACH»
```

The body of a `FOREACH` block can contain any other statements; specifically `FOREACH` statements may be nested. If `ITERATOR` name is specified, an object of the type `xpand2::Iterator` (see API doc for details) is accessible using the specified name. The `SEPARATOR` expression works in the same way as the one for `EXPAND`.

Example:

```
«FOREACH {'A','B','C'} AS c ITERATOR iter SEPARATOR ','»  
  «iter.counter1» : «c»  
«ENDFOREACH»
```

The evaluation of the above statement results in the following text:

```
1 : A,  
2 : B,  
3 : C
```

## IF

The `IF` statement supports conditional expansion. Any number of `ELSEIF` statements are allowed. The `ELSE` block is optional. Every `IF` statement must be closed with an `ENDIF`. The body of an `IF` block can contain any other statement, specifically, `IF` statements may be nested.

```
«IF expression»  
  a sequence of statements  
[ «ELSEIF expression» ]  
  a sequence of statements ]  
[ «ELSE»  
  a sequence of statements ]  
«ENDIF»
```

## PROTECT

Protected Regions are used to mark sections in the generated code that shall not be overridden again by the subsequent generator run. These sections typically contain manually written code.

```
«PROTECT CSTART expression CEND expression ID expression (DISABLE)?»
  a sequence of statements
«ENDPROTECT»
```

The values of CSTART and CEND expressions are used to enclose the Protected Regions marker in the output. They should build valid comment beginning and end strings corresponding to the generated target language (e.g. „/\*“ and „\*/“ for Java). The following is an example for Java:

```
«PROTECT CSTART „/*“ CEND „*/“ ID ElementsUniqueID»
  here goes some content
«ENDPROTECT»
```

The ID is set by the ID expression and must be globally unique (at least for one complete pass of the generator).

Generated target code looks like this:

```
public class Person {
  /*PROTECTED REGION ID(Person) ENABLED START*/
  this pr is enabled, therefore the contents will always be
  preserved. If you want to get the default contents from the
  template you must remove the ENABLED keyword (or even remove
  the whole file :-))
  /*PROTECTED REGION END*/
}
```

Protected regions are generated in enabled state by default. Unless you manually disable them, by removing the ENABLED keyword, they will always be preserved.

If you want the generator to generate disabled protected regions, you need to add the DISABLE keyword inside the declaration:

```
«PROTECT CSTART '/*' CEND '*/' ID this.name DISABLE»
```

## LET

LET lets you specify local variables. :-)

```
«LET expression AS variableName»
  a sequence of statements
«ENDLET»
```

During the expansion of the body of the LET block, the value of the expression is bound to the specified variable. Note that the expression will only be evaluated once, independent from the number of usages of the variable within the LET block. Example:

```
«LET packageName + "." + className AS fqname»  
  the fully qualified name is: «fqname»;  
«ENDLET»
```

## ERROR

The `ERROR` statement aborts the evaluation of the templates by throwing an `XpandException` with the specified message.

```
«ERROR expression»
```

Note that you should use this facility very sparingly, since it is better practice to check for invalid models using constraints on the metamodel, and not in the templates.

## Comments

Comments are only allowed outside of Tags.

```
«REM»  
  text comment  
«ENDREM»
```

Comments may not contain a `REM` tag, this implies that comments are not nestable. A comment may not have a white space between the `REM` keyword and its brackets. Example:

```
«REM»«LET expression AS variableName»«ENDREM»  
  a sequence of statements  
«REM» «variableName.stuff»  
«ENDLET»«ENDREM»
```

## Expression Statement

Expressions support processing of the information provided by the instantiated meta model. *Xpand* provides powerful expressions for selection, aggregation, and navigation. *Xpand* uses the expressions sub language in almost any statement that we have seen so far. The expression statement just evaluates the contained expression and writes the result to the output (using `java.lang.Object`'s `toString()` method). Example:

```
public class «this.name» {
```

All expressions defined by the oAW expressions sub language are also available in *Xpand*. You can invoke imported extensions. (See the *Expressions* and *Extend Language Reference* for more details).

## Controlling generation of white space

If you want to omit the output of superfluous white space you can add a minus character just before any closing bracket. Example:

```
«FILE InterfaceName + ".java"-»  
«IF hasPackage-»  
package «InterfacePackageName»;  
«ENDIF-»  
...  
«ENDFILE»
```

The generated file would start with two new lines (one after the FILE and one after the IF statement) if the minus characters had not been set.

In general: If a statement (or comment) ends with such a minus all preceding whitespace up to the newline character (excluded!) is removed. Additionally all following whitespace including the first newline character (`\r\n` is handled as one character) is also removed.

## Aspect-Oriented Programming in Xpand

Using the workflow engine it is now possible to package (e.g. zip) a written generator and deliver it as a kind of black box. If you want to use such a generator but need to change some small generation stuff, you can make use of the `AROUND` aspects.

```
«AROUND qualifiedDefinitionName (parameterList)? FOR type»  
    a sequence of statements  
«ENDAROUND»
```

`AROUND` lets you add templates in a non-invasive way (you don't need to touch the generator templates). Because aspects are invasive, a template file containing `AROUND` aspects must be wrapped by configuration (see next section).

## Join Point and Point Cut Syntax

AOP is basically about weaving code into different points inside the call graph of a software module. Such points are called *Join Points*. In Xpand there is only one join point so far: a call to a definition.

You specify on which join points the contributed code should be executed by specifying something like a 'query' on all available join points. Such a query is called a *point cut*.

```
«AROUND [pointcut]»  
    do stuff  
«ENDAROUND»
```

A pointcut consists of a fully qualified name, parameter types and the target type.

## Definition Name

The definition name part of a point cut must match the fully qualified name of the join point's definition. Such expressions are case sensitive. The asterisk character is used to specify wildcards.

Some examples:

```
my::Template::definition // definitions with the specified name
org::oaw::* // definitions prefixed with 'org::oaw::'
*Operation* // definitions containing the word 'Operation' in it.
* // all definitions
```

## Parameter Types

The parameters of the definitions we want to add our advice to can also be specified in the point cut. The rule is that the type of the specified parameter must be the same or a super type of the corresponding parameter type (the dynamic type at runtime!) of the definition to be called.

Additionally one can set the wildcard at the end of the parameter list to specify that there might be none or more parameters of any kind.

Some examples:

```
my::Templ::def() // templ def without parameters
my::Templ::def(String s) // templ def with exactly one parameter of
type String
my::Templ::def(String s,*) // templ def with one or more parameters,
where the first parameter is of type String
my::Templ::def(*) // templ def with any number of parameters
```

## Target Type

Finally we have to specify the target type. This is straightforward:

```
my::Templ::def() FOR Object // templ def for any target type
my::Templ::def() FOR Entity // templ def objects of type Entity
```

## Proceeding

Inside an advice you might want to call the underlying definition. This can be done using the implicit variable `targetDef`, which is of the type `xpand2::Definition` and provides an operation `proceed()` which invokes the underlying definition with the original parameters (Note that you might have changed any mutable object in the advice before).

If you want to control what parameters are to be passed to the definition you can use the operation `proceed(Object target, List params)`. There is no type checking here!

Additionally there are some inspection properties (like `name`, `paramTypes`, etc.) available.

## Generator Workflow Component

This section describes the workflow component that is provided to perform the code generation, i.e. run the templates. You should have a basic idea of how the workflow engine works. (see *Workflow Reference*). A simple generator component configuration could look as follows:

```
<component class="oaw.xpand2.Generator2">
  <fileEncoding value="ISO-8859-1"/>
  <metaModel class="oaw.type.emf.EmfMetaModel">
    <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
  </metaModel>
  <expand value="example::Java::all FOR myModel.get(0)"/>

  <!-- aop configuration -->
  <aspects value='example::Aspects1, example::Aspects2' />

  <!-- output configuration -->
  <outlet path='main/src-gen' />
  <outlet name='TO_SRC' path='main/src' overwrite='false' />
  <beautifier class="oaw.xpand2.output.JavaBeautifier"/>
  <beautifier class="oaw.xpand2.output.XmlBeautifier"/>

  <!-- protected regions configuration -->
  <prSrcPathes value="main/src"/>
  <prDefaultExcludes value="false"/>
  <prExcludes value="*.xml"/>
</component>
```

Let's go through the different properties one by one.

### Main configuration

The first thing to note, is that the qualified Java name of the component is `org.openarchitectureware.xpand2.Generator2`. One can use the shortcut `oaw` instead of a preceding `org.openarchitectureware`. The workflow engine will resolve it.

### Encoding

For *Xpand* it's important to have the file encoding in mind, because of the guillemet characters used to delimit keywords and property access. The `fileEncoding` property specifies the file encoding to use for reading the templates, reading the protected regions and writing the generated files. This property defaults to the default file encoding of your JVM.

## Metamodel

The property `metaModel` is used to tell the generator engine on which metamodels the xpanse templates should be evaluated. One can specify more than one metamodel here. Metamodel implementations are required by the expression framework (see *Expression Language Reference*) used by Xpanse2. In the example above we configured the Ecore metamodel using the *EMFMetaModel* implementation shipped with the core part of the *openArchitectureWare* 4 release.

A mandatory configuration is the `expand` property. It expects a syntax similar to that of the `EXPAND` statement (described above). The only difference is that we omit the `EXPAND` keyword because we write it as the property's name. Examples:

```
<expand value="Template::define FOR mySlot"/>
```

or:

```
<expand value="Template::define('foo') FOREACH {mySlot1,mySlot2}"/>
```

The expressions are evaluated using the workflow context. Each slot is mapped to a variable. For the examples above the workflow context needs to contain elements in the slots `'mySlot'`, `'mySlot1'` and `'mySlot2'`. It's also possible to specify some complex expressions here. If, for instance, the slot `myModel` contains a collection of model elements one could write:

```
<expand value="Template::define FOREACH myModel.typeSelect(Entity)"/>
```

This selects all elements of type *Entity* contained in the collection stored in the `myModel` slot.

## Output configuration

The second mandatory configuration is the specification of so called outlets (a concept borrowed from androMDA ;-)). Outlets are responsible for writing the generated files to disk. Example:

```
<component class="oaw.xpanse2.Generator2">
  ...
  <outlet path='main/src-gen' />
  <outlet name='TO_SRC' path='main/src' overwrite='false' />
  ...
</component>
```

In the example there are two outlets configured. The first one has no name and is therefore handled as the default outlet. Default outlets are triggered by omitting an outlet name:

```
<<FILE 'test/note.txt'>>
# this goes to the default outlet
<<ENDFILE>>
```

The configured base path is 'main/src-gen', so the file from above would go to 'main/src-gen/test/note.txt'.

The second outlet has a name ('TO\_SRC') specified. Additionally the flag `overwrite` is set to false (defaults to true). The following Xpand fragment

```
«FILE 'test/note.txt' TO_SRC»  
# this goes to the TO_SRC outlet  
«ENDFILE»
```

would cause the generator to write the contents to 'main/src/test/note.txt' if the file does not already exist (the `overwrite` flag).

Another option called `append` (defaults to false) causes the generator to append the generated text to an existing file. If `overwrite` is set to false this flag has no effect.

## Beautifier

Beautifying the generated code is a good idea. It's very important, that generated code look good, because developers should be able to understand it. On the other hand template files should look good, too. It is thus best practice to write nice looking template files and not to care how the generated code looks – and then you run a beautifier over the generated code to fix that problem. Of course, if a beautifier is not available, or if white space has syntactical meaning (as in Python), you would have to write your templates with that in mind (using the minus character before closing brackets as described in a preceding section).

The *Xpand* workflow component can be configured with multiple beautifiers:

```
<beautifier  
  class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>  
<beautifier  
  class="org.openarchitectureware.xpand2.output.XmlBeautifier"/>
```

These are the two beautifiers delivered with *Xpand*. If you want to use your own beautifier, you would just need to implement the following Java interface:

```
package org.openarchitectureware.xpand2.output;  
  
public interface PostProcessor {  
  public void beforeWriteAndClose(FileHandle handle);  
}
```

The `beforeWriteAndClose` method is called for each `ENDFILE` statement.

## JavaBeautifier

The `JavaBeautifier` is based on `Jalopy` and provides base beautifying for Java files.

## JavaBeautifier2

The Java beautifier (implemented by Karsten Klein, hybrid-labs) is based on Jalopy and offers two options.

The option `format` (defaults to `true`) specifies, whether the Java code should be pretty printed using Jalopy.

The option `organizeImports` (defaults to `true`) specifies, whether the beautifier should substitute fully qualified class names (as long as there is no conflict) with the short name and the corresponding import statement.

## XmlBeautifier

The XmlBeautifier is based on dom4j and provides a single option `fileExtensions` (defaults to `„.xml, .xsl, .wsdd, .wsdl“`) used to specify which files should be pretty printed.

## Protected Region Configuration

Finally you need to configure the protected region resolver, if you want to use protected regions.

```
<prSrcPaths value="main/src"/>
<prDefaultExcludes value="false"/>
<prExcludes value="*.xml"/>
```

The `prSrcPaths` property points to a comma-separated list of directories. The protected region resolver will scan these directories for files containing activated protected regions.

There are several file names which are excluded by default:

```
RCS,SCCS,CVS,CVS.adm,RCSLOG,cvslog.*,tags,TAGS,.make.state
.nse_depinfo,*~,#*,.##*,',*',_$$*,*$*,*.old,*.bak,*.BAK,*.orig,*.rej,
.del-*,*.a,*.olb,*.o,*.obj,*.so,*.exe,*.Z,*.elc,*.ln,core
```

If you don't want to exclude any of these, you must set `prDefaultExcludes` to `false`.

```
<prDefaultExcludes value="false"/>
```

If you want to add additional excludes, you should use the `prExcludes` property.

```
<prExcludes value="*.xml,*.hbm"/>
```

**Note:** It's bad practice to mix generated and non-generated code in one artefact.. Instead of using protected regions, you should try to leverage the target language's extension features (inheritance, inclusion, references, etc.) wherever possible. It is very rare that the use of protected regions is an appropriate solution.

## About our Sponsors

**itemis GmbH & Co. KG** is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

**b+m** is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering as well as the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hannover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

**oose Innovative Informatik GmbH** offers coaching, consulting and training in all areas of software engineering. The main focus of their activities is on software architecture, requirements engineering and project-management. We at *oose* have first-hand information and experience, because our staff take part actively with others in current trends, standards and innovations. Our staff support this and regularly pass their know-how on by writing and publishing books or speaking at conferences, etc. Within the OMG, oose collaborates actively on the specifications of the UML and also the SysML.

**MID Enterprise Software Solutions GmbH** is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production. The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.