Xpand Documentation

Xpand Documentation

Chapter 1. *Xpand / Xtend / Check* Reference

1. Introduction

The *Xpand* generator framework provides textual languages, that are useful in different contexts in the MDSD process (e.g. checks, extensions, code generation, model transformation). Each language (*Check*, *Xtend*, and *Xpand*) is built up on a common expression language and type system. Therefore, they can operate on the same models, metamodels and meta-metamodels and you do not need to learn the syntax again and again, because it is always the same.

The expressions framework provides a uniform abstraction layer over different meta-meta-models (e.g. EMF Ecore, Eclipse UML, JavaBeans, XML Schema etc.). Additionally, it offers a powerful, statically typed expressions language, which is used in the various textual languages.

2. Type System

The abstraction layer on API basis is called a type system. It provides access to built-in types and different registered metamodel implementations. These registered metamodel implementations offer access to the types they provide. The first part of this documentation describes the type system. The expression sub-language is described afterwards in the second part of this documentation. This differentiation is necessary because the type system and the expression language are two different things. The type system is a kind of reflection layer, that can be extended with metamodel implementations. The expression language defines a concrete syntax for executable expressions, using the type system.

The Java API described here is located in the org.eclipse.xpand.type package and is a part of the subproject core.expressions.

2.1. Types

Every object (e.g. model elements, values, etc.) has a type. A type contains properties and operations. In addition it might inherit from other types (multiple inheritance).

2.1.1. Type Names

Types have a simple name (e.g. `String`) and an optional namespace used to distingish between two types with the same name (e.g. `my::metamodel`). The delimiter for name space fragments is a double colon "`::`". A fully qualified name looks like this:

```
my::fully::qualified::MetaType
```

The namespace and name used by a specific type is defined by the corresponding `MetaModel` implementation. The `EmfMetaModel`, for instance, maps `EPackages` to namespace and `EClassifiers` to names. Therefore, the name of the Ecore element `EClassifier` is called:

```
ecore::EClassifier
```

If you do not want to use namespaces (for whatever reason), you can always implement your own metamodel and map the names accordingly.

2.1.2. Collection Type Names

The built-in type system also contains the following collection types: `Collection`, `List` and `Set`. Because the expressions language is statically type checked and we do not like casts and `ClassCastExceptions`, we introduced the concept of *parameterized types*. The type system does not support full featured generics, because we do not need them.

The syntax is:

```
Collection[my::Type]
List[my::Type]
```

```
Set[my::Type]
```

2.1.3. Features

Each type offers features. The type (resp. the metamodel) is responsible for mapping the features. There are three different kinds of features:

- Properties
- Operations
- Static properties

Properties are straight forward: They have a name and a type. They can be invoked on instances of the corresponding type. The same is true for *Operations*. But in contrast to properties, they can have parameters. *Static properties* are the equivalent to enums or constants. They must be invoked statically and they do not have parameters.

2.2. Built-In Types

As mentioned before, the expressions framework has several built-in types that define operations and properties. In the following, we will give a rough overview of the types and their features. We will not document all of the operations here, because the built-in types will evolve over time and we want to derive the documentation from the implementation (model-driven, of course). For a complete reference, consult the generated API documentation (http://www.openarchitectureware.org/api/built-ins/).

2.2.1. `Object`

`Object` defines a couple of basic operations, like `equals()`. Every type has to extend `Object`.

2.2.2. `Void`

The `Void` type can be specified as the return type for operations, although it is not recommended, because whenever possible expressions should be free of side effects whenever possible.

2.2.3. Simple types (Data types)

The type system doesn't have a concept data type. Data types are just types. As in OCL, we support the following types: `String`, `Boolean`, `Integer`, `Real`.

- `String`: A rich and convenient `String` library is especially important for code generation. The type system supports the '+' operator for concatenation, the usual `java.lang.String` operations (`length()`, etc.) and some special operations (like `toFirstUpper()`, `toFirstLower()`, regular expressions, etc. often needed in code generation templates).
- `Boolean`: `Boolean` offers the usual operators (Java syntax): &&, ||, !, etc.
- `Integer` and `Real`: `Integer` and `Real` offer the usual compare operators (<,>,<=,>=) and simple arithmetics (+,-,*,/). Note that *`Integer` extends `Real`!*

2.2.4. Collection types

The type system has three different Collection types. `Collection` is the base type, it provides several operations known from `java.util.Collection`. The other two types (`List`, `Set`) correspond to their java.util equivalents, too.

2.2.5. Type system types

The type system describes itself, hence, there are types for the different concepts. These types are needed for reflective programming. To avoid confusion with metatypes with the same name (it is not unusual to have a metatype called `Operation`, for instance) we have prefixed all of the types with the namespace `xpand`. We have:

- `xpand2::Type`
- `xpand2::Feature`
- `xpand2::Property`
- `xpand2::StaticProperty`

- `xpand2::Operation`

2.3. Metamodel Implementations (also known as Meta-Metamodels)

By default, the type system only knows the built-in types. In order to register your own metatypes (e.g. `Entity` or `State`), you need to register a respective metamodel implementation with the type system. Within a metamodel implementation the *Xpand* type system elements (Type, Property, Operation) are mapped to an arbitrary other type system (Java reflections, Ecore or XML Schema).

2.3.1. Example JavaMetaModel

For instance, if you want to have the following JavaBean act as a metatype (i.e. your model contains instances of the type):

```
public class Attribute {
   private String name;
   private String type;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getType() {
      return type;
   }
   public void setType(String type) {
      this.type = type;
   }
}
```

You need to use the `JavaMetaModel` implementation which uses the ordinary Java reflection layer in order to map access to the model.

So, if you have the following expression in e.g. *Xpand*:

```
myattr.name.toFirstUpper()
```

and `myattr` is the name of a local variable pointing to an instance of `Attribute`. The *Xpand* type system asks the metamodel implementations, if they 'know' a type for the instance of Attribute. If you have the `JavaMetaModel` registered it will return an `xpand2::Type` which maps to the underlying Java class. When the type is asked if it knows a property '`name`', it will inspect the Java class using the Java reflection API.

The JavaMetaModel implementation shipped with *Xpand* can be configured with a strategy [GOF95-Pattern] in order to control or change the mapping. For instance, the JavaBeansStrategy maps getter and setter methods to simple properties, so we would use this strategy for the example above.

2.3.2. Eclipse IDE MetaModelContributors

You should know that for each `Metamodel` implementation you use at runtime, you need to have a so called `MetamodelContributor` extension for the plugins to work with. If you just use one of the standard metamodel implementations (EMF, UML2 or Java) you don't have to worry about it, since *Xpand* is shipped with respective MetamodelContributors (see the corresponding docs for details). If you need to implement your own `MetamodelContributor` you should have a look at the Eclipse plug-in reference doc.

2.3.3. Configuring Metamodel implementations with the workflow

You need to configure your *Xpand* language components with the respective metamodel implementations.

A possible configuration of the `Xpand2` generator component looks like this:

```
<component class="org.eclipse.xpand2.Generator">
   <metaModel class="org.eclipse.type.emf.EmfMetaModel">
      <metaModelPackage value="my.generated.MetaModel1Package"/>
   </metaModel>
   <metaModel class="org.eclipse.type.emf.EmfMetaModel">
      <metaModelFile value="my/java/package/metamodel2.ecore"/>
   </metaModel>
   ...
```

```
</component>
```

In this example the `EmfMetaModel` implementation is configured two times. This means that we want to use two metamodels at the same time, both based on EMF. The metaModelPackage property is a property that is specific to the `EmfMetaModel` (located in the `core.emftools` project). It points to the generated `EPackages` interface. The second meta model is configured using the Ecore file. You do no need to have a generated Ecore model for *Xpand* in order to work. The `EmfMetaModel` works with dynamic EMF models just as it works with generated EMF models.

2.4. Using different Metamodel implementations (also known as Meta-Metamodels)

With Xpad you can work on different kinds of Model representations at the same time in a transparent manner. One can work with EMF models, XML DOM models, and simple JavaBeans in the same *Xpand* template. You just need to configure the respective MetaModel implementations.

If you want to do so you need to know how the type lookup works. Let us assume that we have an EMF metamodel and a model based on some Java classes. Then the following would be a possible configuration:

```
<component class="org.eclipse.xpand2.Generator">
   <metaModel class="org.eclipse.internal.xtend.type.impl.java.JavaMetaModel"/>
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
      <metaModelFile value="my/java/package/metamodel.ecore"/>
   </metaModel>

   ...
</component>
```

When the runtime needs to access a property of a given object, it asks the metamodels in the configured order. Let us assume that our model element is an instance of the Java type `org.eclipse.emf.ecore.EObject` and it is a dynamic instance of an EMF EClass `MyType`.

We have *three* Metamodels:

1. Built-Ins (always the first one)
2. JavaMetaModel
3. EMFMetaModel - metamodel.ecore

The first one will return the type `Object` (not `java.lang.Object` but `Object` of *Xpand*). At this point the type `Object` best fits the request, so it will act as the desired type.

The second metamodel returns a type called `org::eclipse::emf::ecore::EObject` The type system will check if the returned type is a specialization of the current 'best-fit' type (`Object`). It is, because it extends `Object` (Every metatype has to extend `Object`). At this time the type system assumes `org::eclipse::emf::ecore::EObject` to be the desired type.

The third metamodel will return `metamodel::MyType` which is the desired type. But unfortunately it doesn't extend `org::eclipse::emf::ecore::EObject` as it has nothing to do with those Java types. Instead it extends `emf::EObject` which extends `Object`.

We need to swap the configuration of the two metamodels to get the desired type.

```
<component class="org.eclipse.xpand2.Generator">
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
      <metaModelFile value="my/java/package/metamodel.ecore"/>
   </metaModel>
   <metaModel class="org.eclipse.internal.xtend.type.impl.java.JavaMetaModel"/>

   ...
</component>
```

3. Expressions

The expression sub-language is a syntactical mixture of Java and OCL. This documentation provides a detailed description of each available expression. Let us start with some simple examples.

Accessing a property:

```
myModelElement.name
```

Accessing an operation:

```
myModelElement.doStuff()
```

simple arithmetic:

```
1 + 1 * 2
```

boolean expressions (just an example:-)):

```
!('text'.startsWith('t') && ! false)
```

3.1. Literals and special operators for built-in types

There are several literals for built-in types:

3.1.1. `Object`

There are naturally no literals for object, but we have two operators:

equals:

```
obj1 == obj2
```

not equals:

```
obj1 != obj2
```

3.1.2. `Void`

The only possible instance of `Void` is the `null` reference. Therefore, we have one literal:

```
null
```

3.1.3. Type literals

The literal for types is just the name of the type (no '.class' suffix, etc.). Example:

```
String // the type string
my::special::Type // evaluates to the type 'my::special::Type'
```

3.1.4. StaticProperty literals

The literal for static properties (aka enum literals) is correlative to type literals:

```
my::Color::RED
```

3.1.5. `String`

There are two different literal syntaxes (with the same semantics):

```
'a String literal'
"a String literal" // both are okay
```

For Strings the expression sub-language supports the plus operator that is overloaded with concatenation:

```
'my element '+ ele.name +' is really cool!'
```

Note, that multi-line Strings are supported.

3.1.6. `Boolean`

The boolean literals are:

```
true
false
```

Operators are:

```
true && false // AND
true || false // OR
! true        // NOT
```

3.1.7. `Integer` and `Real`

The syntax for integer literals is as expected:

```
// integer literals
3
57278
// real literals
3.0
0.75
```

Additionally, we have the common arithmetic operators:

```
3 + 4  // addition
4 - 5  // subtraction
2 * 6  // multiplication
3 / 64 // divide
// Unary minus operator
- 42
- 47.11
```

Furthermore, the well known compare operators are defined:

```
4 > 5 // greater than
4 < 5 // smaller than
4 >= 23 // greater equals than
4 <= 12 // smaller equals than
```

3.1.8. Collections

There is a literal for lists:

```
{1,2,3,4} // a list with four integers
```

There is no other special concrete syntax for collections. If you need a set, you have to call the `toSet()` operation on the list literal:

```
{1,2,4,4}.toSet() // a set with 3(!) integers
```

3.2. Special Collection operations

Like OCL, the *Xpand* expression sub-language defines several special operations on collections. However, those operations are not members of the type system, therefore you cannot use them in a reflective manner.

3.2.1. `select`

Sometimes, an expression yields a large collection, but one is only interested in a special subset of the collection. The expression sub-language has special constructs to specify a selection out of a specific collection. These are the `select` and `reject` operations. The select specifies a subset of a collection. A `select` is an operation on a collection and is specified as follows:

```
collection.select(v | boolean-expression-with-v)
```

`select` returns a sublist of the specified collection. The list contains all elements for which the evaluation of boolean-expression-with-v results is `true`. Example:

```
{1,2,3,4}.select(i | i >= 3) // returns {3,4}
```

3.2.2. `typeSelect`

A special version of a select expression is `typeSelect`. Rather than providing a boolean expression a class name is here provided.

```
collection.typeSelect(classname)
```

`typeSelect` returns that sublist of the specified collection, that contains only objects which are an instance of the specified class (also inherited).

3.2.3. `reject`

The `reject` operation is similar to the `select` operation, but with `reject` we get the subset of all the elements of the collection for which the expression evaluates to `false`. The `reject` syntax is identical to the `select` syntax:

```
collection.reject(v | boolean-expression-with-v)
```

Example:

```
{1,2,3,4}.reject(i | i >= 3) // returns {1,2}
```

3.2.4. `collect`

As shown in the previous section, the `select` and `reject` operations always result in a sub-collection of the original collection. Sometimes one wants to specify a collection which is derived from another collection, but which contains objects that are not in the original collection (it is not a sub-collection). In such cases, we can use a `collect` operation. The `collect` operation uses the same syntax as the `select` and `reject` and is written like this:

```
collection.collect(v | expression-with-v)
```

`collect` again iterates over the target collection and evaluates the given expression on each element. In contrast to `select`, the evaluation result is collected in a list. When an iteration is finished the list with all results is returned. Example:

```
namedElements.collect(ne | ne.name) // returns a list of strings
```

3.2.5. Shorthand for `collect` (and more than that)

As navigation through many objects is very common, there is a shorthand notation for collect that makes the expressions more readable. Instead of

```
self.employee.collect(e | e.birthdate)
```

one can also write:

```
self.employee.birthdate
```

In general, when a property is applied to a collection of Objects, it will automatically be interpreted as a `collect` over the members of the collection with the specified property.

The syntax is a shorthand for `collect`, if the feature does not return a collection itself. But sometimes we have the following:

```
self.buildings.rooms.windows // returns a list of windows
```

This syntax works, but one cannot express it using the `collect` operation in an easy way.

3.2.6. `forAll`

Often a boolean expression has to be evaluated for all elements in a collection. The `forAll` operation allows specifying a Boolean expression, which must be `true`for all objects in a collection in order for the `forAll` operation to return `true`:

```
collection.forAll(v | boolean-expression-with-v)
```

The result of `forAll` is `true` if `boolean-expression-with-v` is `true` for all the elements contained in a collection. If `boolean-expression-with-v` is `false` for one or more of the elements in the collection, then the `forAll` expression evaluates to `false`.

Example:

```
{3,4,500}.forAll(i | i < 10) // evaluates to false (500 < 10 is false)
```

3.2.7. `exists`

Often you will need to know whether there is at least one element in a collection for which a boolean is `true`. The exists operation allows you to specify a Boolean expression which must be `true` for at least one object in a collection:

```
collection.exists(v | boolean-expression-with-v)
```

The result of the exists operation is `true` if `boolean-expression-with-v` is `true` for at least one element of collection. If the `boolean-expression-with-v` is `false` for all elements in collection, then the complete expression evaluates to `false`.

Example:

```
{3,4,500}.exists(i | i < 10) // evaluates to true (e.g. 3 < 10 is true)
```

3.2.8. `sortBy`[1]

If you want to sort a list of elements, you can use the higher order function `sortBy` . The list you invoke the `sortBy` operation on, is sorted by the results of the given expression.

Example:

```
myListOfEntity.sortBy(entity | entity.name)
```

In the example the list of entities is sorted by the name of the entities. Note that there is no such `Comparable` type in *Xpand*. If the values returned from the expression are instances of `java.util.Comparable` the `compareTo` method is used, otherwise `toString()` is invoked and the the result is used.

More Examples â## all the following expressions return `true`:

```
{'C','B','A'}.sortBy(e | e) == {'A','B','C'}
{'AAA','BB','C'}.sortBy(e | e.length) == {'C','BB','AAA'}
{5,3,1,2}.sortBy(e | e) == {1,2,3,5}
{5,3,1,2}.sortBy(e | e - 2 * e) == {5,3,2,1}
...
```

3.3. `if` expression

There are two different forms of conditional expressions. The first one is the so-called *if expression*. Syntax:

```
condition ? thenExpression : elseExpression
```

Example:

```
name != null ? name : 'unknown'
```

3.4. `switch` expression

The other one is called *switch expression*. Syntax:

```
switch (expression) {
   (case expression : thenExpression)*
   default : catchAllExpression
}
```

The default part is mandatory, because `switch` is an expression, therefore it needs to evaluate to something in any case. Example:

```
switch (person.name) {
   case 'Hansen' : 'Du kanns platt schnacken'
   default : 'Du kanns mi nech verstohn!'
```

[1] since 4.1.2

```
}
```

There is an abbreviation for *Boolean* expressions:

```
switch {
   case booleanExpression : thenExpression
   default : catchAllExpression
}
```

3.5. Chain expression

Expressions and functional languages should be free of side effects as far as possible. But sometimes there you need invocations that do have side effects. In some cases expressions even don not have a return type (i.e. the return type is `Void`). If you need to call such operations, you can use the chain expression. Syntax:

```
anExpr ->
anotherExpr ->
lastExpr
```

Each expression is evaluated in sequence, but only the result of the last expression is returned. Example:

```
pers.setName('test') ->
pers
```

This chain expression will set the `name` of the person first, before it returns the person object itself.

3.6. `create` expression

The `create` expression is used to instantiate new objects of a given type:

```
new TypeName
```

3.7. `let` expression

The `let` expression lets you define local variables. Syntax is as follows:

```
let v = expression : expression-with-v
```

This is especially useful together with a chain- and a create expressions. Example:

```
let p = new Person :
   p.name('John Doe') ->
   p.age(42) ->
   p.city('New York') ->
   p
```

3.8. 'GLOBALVAR' expression

Sometimes you don't want to pass everything down the call stack by parameter. Therefore, we have the `GLOBALVAR` expression. There are two things you need to do, to use global variables.

3.8.1. Using GLOBALVARS to configure workflows

Each workflow component using the expression framework (*Xpand*, *Check* and *Xtend*) can be configured with global variables. Here is an example:

```
<workflow>
   .... stuff
   <component class="org.eclipse.xpand2.Generator">
      ... usual stuff (see ref doc)
      <globalVarDef name="MyPSM" value="slotNameOfPSM"/>
      <globalVarDef name="ImplClassSuffix" value="'Impl'"/>
   </component>
</workflow>
```

If you have injected global variables into the respective component, you can call them using the following syntax:

```
GLOBALVAR ImplClassSuffix
```

Note, we don't have any static type information. Therefore `Object` is assumed. So, you have to down cast the global variable to the intended type:

```
((String) GLOBALVAR ImplClassSuffix)
```

It is good practice to type it once, using an Extension and then always refer to that extension:

```
String implClassSuffix() : GLOBALVAR ImplClassSuffix;
// usage of the typed global var extension
ImplName(Class c) :
   name+implClassSuffix();
```

3.9. Multi methods (multiple dispatch)

The expressions language supports multiple dispatching . This means that when there is a bunch of overloaded operations, the decision which operation has to be resolved is based on the dynamic type of all parameters (the implicit '`this`' included).

In Java only the dynamic type of the '`this`' element is considered, for parameters the static type is used. (this is called single dispatch)

Here is a Java example:

```
class MyClass {
   boolean equals(Object o) {
      if (o instanceof MyClass) {
         return equals((MyClass)o);
      }
      return super.equals(o);
   }
   boolean equals(MyType mt) {
      //implementation...
   }
}
```

The method `equals(Object o)` would not have to be overwritten, if Java would support multiple dispatch.

3.10. Casting

The expression language is statically type checked. Although there are many concepts that help the programmer to have really good static type information, sometimes. one knows more about the real type than the system. To explicitly give the system such an information casts are available. *Casts are 100% static, so you do not need them, if you never statically typecheck your expressions!*

The syntax for casts is very Java-like:

```
((String)unTypedList.get(0)).toUpperCase()
```

4. *Check*

Xpand also provides a language to specify constraints that the model has to fulfill in order to be correct. This language is very easy to understand and use. Basically, it is built around the expression syntax that has been discussed in detail in the previous section. Constraints specified in the *Check* language have to be stored in files with the file extension `.chk` . Furthermore, these files have to be on the Java classpath, of course, in order to be found. Let us look at an example, in order to understand, what these constraints look like and what they do:

```
import data;
context Attribute ERROR
  "Names have to be more than one character long." :
  name.length > 1;
```

Now, let us look at the example line by line:

1. First, the metamodel has to be imported.
2. Then, the context is specified for which the constraint applies. In other words, after the `context` keyword, we put the name of the metaclass that is going to be checked by the constraint. Then, there follows either `ERROR` or `WARNING`, These keywords specify what kind of action will be taken in case the constraint fails:

Table 1.1. Types of action for *Check* constraints

`WARNING`	If the constraint fails, the specified message is printed, but the workflow execution is not stopped.
`ERROR`	If the constraint fails, the specified message is printed and all further processing is stopped.

3. Now, the message that is put in case that the constraint fails is specified as a string. It is possible to include the value of attributes or the return value of functions into the message in order to make the message more clear. For example, it would be possible to improve the above example by rewriting it like this:

```
import data;
context Attribute ERROR
  "Name of '" + name + "too short. Names have to be more than one character long." :
  name.length > 1;
```

4. Finally, there is the condition itself, which is specified by an expression, which has been discussed in detail in the previous section. If this expression is `true`, the constraint is fulfilled.

> Please always keep in mind that the message that is associated with the constraint is printed, if the condition of the constraint is `false`! Thus, if the specified constraint condition is `true`, nothing will be printed out and the constraint will be fulfilled.

4.1. Guard Conditions

The *Check* language of *Xpand* also provides so called . These conditions allow to apply a check constraint only to model elements that meet certain criteria. Specifying such a guard condition is done by adding an *if* clause to the check constraint. The *if* clause has to be added after the *context* clause as demonstrated by the following example:

```
import data;
context Attribute if name.length > 1 ERROR
  "Attribute names have to start with an 'a'" :
  name.startsWith("a");
```

5. *Xtend*

Like the expressions sublanguage that summarizes the syntax of expressions for all the other textual languages delivered with the *Xpand* framework, there is another commonly used language called *Xtend*.

This language provides the possibility to define rich libraries of independent operations and non-invasive metamodel extensions based on either Java methods or*Xtend* expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework.

5.1. Extend files

An extend file must reside in the Java class path of the used execution context. Additionally it is file extension must be `*.ext`. Let us have a look at an extend file.

```
import my::metamodel;extension other::ExtensionFile;

/**
  * Documentation
  */
anExpressionExtension(String stringParam) :
  doingStuff(with(stringParam))
;

/**
  * java extensions are just mappings
  */
String aJavaExtension(String param) : JAVA
  my.JavaClass.staticMethod(java.lang.String)
;
```

The example shows the following statements:

1. import statements

2. extension import statements
3. expression or java extensions

5.2. Comments

We have single- and multi-line comments. The syntax for single line comments is:

```
// my comment
```

Multi line comments are written like this:

```
/* My multi line comment */
```

5.3. Import Statements

Using the import statement one can import name spaces of different types.(see expressions framework reference documentation).

Syntax is:

```
import my::imported::namespace;
```

Extend does not support static imports or any similar concept. Therefore, the following is incorrect syntax:

```
import my::imported::namespace::*; // WRONG! import my::Type; // WRONG!
```

5.4. Extension Import Statement

You can import another extend file using the extension statement. The syntax is:

```
extension fully::qualified::ExtensionFileName;
```

Note, that no file extension (`*.ext`) is specified.

5.4.1. Reexporting Extensions

If you want to export extensions from another extension file together with your local extensions, you can add the keyword 'reexport' to the end of the respective extension import statement.

```
extension fully::qualified::ExtensionFileName reexport;
```

5.5. Extensions

The syntax of a simple expression extension is as follows:

```
ReturnType extensionName(ParamType1 paramName1, ParamType2...): expression-using-params;
```

Example:

```
String getterName(NamedElement ele) : 'get'+ele.name.firstUpper();
```

5.5.1. Extension Invocation

There are two different ways of how to invoke an extension. It can be invoked like a function:

```
getterName(myNamedElement)
```

The other way to invoke an extension is through the "member syntax":

```
myNamedElement.getterName()
```

For any invocation in member syntax, the target expression (the member) is mapped to the first parameter. Therefore, both syntactical forms do the same thing.

It is important to understand that extensions are not members of the type system, hence, they are not accessible through reflection and you cannot specialize or overwrite operations using them.

The expression evaluation engine first looks for an appropriate operation before looking for an extension, in other words operations have higher precedence.

5.5.2. Type Inference

For most extensions, you do not need to specify the return type, because it can be derived from the specified expression. The special thing is, that the static return type of such an extension depends on the context of use.

For instance, if you have the following extension

```
asList(Object o): {o};
```

the invocation of

```
asList('text')
```

has the static type List[String]. This means you can call

```
asList('text').get(0).toUpperCase()
```

The expression is statically type safe, because its return type is derived automatically.

There is always a return value, whether you specify it or not, even if you specify explicitly '`Void`'.

See the following example.

```
modelTarget.ownedElements.addAllNotNull(modelSource.contents.duplicate())
```

In this example duplicate() dispatches polymorphically. Two of the extensions might look like:

```
Void duplicate(Realization realization):
   realization.Specifier().duplicate()->
   realization.Realizer().duplicate()
;

create target::Class duplicate(source::Class):
   ...
;
```

If a '`Realization`' is contained in the '`contents`' list of '`modelSource`', the '`Realizer`' of the '`Realization`' will be added to the '`ownedElements`' list of the '`modelTarget`'. If you do not want to add in the case that the contained element is a 'Realization' you might change the extension to:

```
Void duplicate(Realization realization):
   realization.Specifier().duplicate()->
   realization.Realizer().duplicate() ->
   {}
;
```

5.5.3. Recursion

There is only one exception: For recursive extensions the return type cannot be inferred, therefore you need to specify it explicitly:

```
String fullyQualifiedName(NamedElement n) : n.parent == null ? n.name :
   fullyQualifiedName(n.parent)+'::'+n.name
;
```

Recursive extensions are non-deterministic in a static context, therefore, it is necessary to specify a return type.

5.5.4. Cached Extensions

If you call an extension without side effects very often, you would like to cache the result for each set of parameters, in order improve the performance. You can just add the keyword '`cached`' to the extension in order to achieve this:

```
cached String getterName(NamedElement ele) :
   'get'+ele.name.firstUpper()
;
```

The `getterName` will be computed only once for each `NamedElement`.

5.5.5. Private Extensions

By default all extensions are public, i.e. they are visible from outside the extension file. If you want to hide extensions you can add the keyword 'private' in front of them:

```
private internalHelper(NamedElement ele) :
   // implementation....
;
```

5.6. Java Extensions

In some rare cases one does want to call a Java method from inside an expression. This can be done by providing a Java extension:

```
Void myJavaExtension(String param) :
   JAVA my.Type.staticMethod(java.lang.String)
;
```

The signature is the same as for any other extension. The implementation is redirected to a public static method in a Java class.

Its syntax is:

```
JAVA fully.qualified.Type.staticMethod(my.ParamType1,
                                       my.ParamType2,
                                       ...)
;
```

Note that you cannot use any imported namespaces. You have to specify the type, its method and the parameter types in a fully qualified way.

Example:

If you have defined the following Java extension:

```
Void dump(String s) :
   JAVA my.Helper.dump(java.lang.String)
;
```

and you have the following Java class:

```
package my;

public class Helper {
   public final static void dump(String aString) {
      System.out.println(aString);
   }
}
```

the expressions

```
dump('Hello world!')
'Hello World'.dump()
```

both result are invoking the Java method void dump(String aString)

5.7. Create Extensions (Model Transformation)

Since Version 4.1 the *Xtend* language supports additional support for model transformation. The new concept is called *create extension* and it is explained a bit more comprehensive as usual.

Elements contained in a model are usually referenced multiple times. Consider the following model structure:

A package P contains two classes C1 and C2. C1 contains a reference R of type C2 (P also references C2).

We could write the following extensions in order to transform an Ecore (EMF) model to our metamodel (Package, Class, Reference).

```
toPackage(EPackage x) :
   let p = new Package :
      p.ownedMember.addAll(x.eClassifiers.toClass()) ->
      p;

toClass(EClass x) :
   let c = new Class :
      c.attributes.addAll(x.eReferences.toReference()) ->
      c;

toReference(EReference x) :
   let r = new Reference :
      r.setType(x.eType.toClass()) ->
      r;
```

For an Ecore model with the above structure, the result would be:

```
    P
   / \
  C1 C2
  |
  R - C2
```

What happened? The C2 class has been created 2 times (one time for the package containment and another time for the reference R that also refers to C2). We can solve the problem by adding the 'cached' keyword to the second extension:

```
cached toClass(EClass x) :
   let c = new Class :
      c.attributes.addAll(c.eAttributes.toAttribute()) ->
      c;
```

The process goes like this:

1. start create P
   a. start create C1 (contained in P)
      i. start create R (contained in C1)
         A. start create C2 (referenced from R)
         B. end (result C2 is cached)
      ii. end R
   b. end C1
   c. start get cached C2 (contained in P)
2. end P

So this works very well. We will get the intended structure. But what about circular dependencies? For instance, C2 could contain a Reference R2 of type C1 (bidirectional references):

The transformation would occur like this:

1. start create P
   a. start create C1 (contained in P)
      i. start create R (contained in C1)
         A. start create C2 (referenced from R)
            I. start create R2 (contained in C2)
               1. start create C1 (referenced from R1)... OOPS!

C1 is already in creation and will not complete until the stack is reduced. Deadlock! The problem is that the cache caches the return value, but C1 was not returned so far, because it is still in construction. The solution: create extensions

The syntax is as follows:

```
create Package toPackage(EPackage x) :
   this.classifiers.addAll(x.eClassifiers.toClass());

create Class toClass(EClass x) :
   this.attributes.addAll(x.eReferences.toReference());

create Reference toReference(EReference x) :
   this.setType(x.eType.toClass());
```

This is not only a shorter syntax, but it also has the needed semantics: The created model element will be added to the cache before evaluating the body. The return value is always the reference to the created and maybe not completely initialized element.

5.8. Calling Extensions From Java

The previous section showed how to implement Extensions in Java. This section shows how to call Extensions from Java.

```
// setup
XtendFacade f = XtendFacade.create("my::path::MyExtensionFile");

// use
f.call("sayHello",new Object[]{"World"});
```

The called extension file looks like this:

```
sayHello(String s) :
   "Hello " + s;
```

This example uses only features of the BuiltinMetaModel, in this case the "+" feature from the StringTypeImpl.

Here is another example, that uses the `JavaBeansMetaModel` strategy. This strategy provides as additional feature: the access to properties using the getter and setter methods.

For more information about type systems, see the *Expressions* reference documentation.

We have one JavaBean-like metamodel class:

```
package mypackage;
public class MyBeanMetaClass {
   private String myProp;
   public String getMyProp() { return myProp; }
   public void setMyProp(String s) { myProp = s;}
}
```

in addition to the built-in metamodel type system, we register the `JavaMetaModel` with the `JavaBeansStrategy` for our facade. Now, we can use also this strategy in our extension:

```
// setup facade

XtendFacade f = XtendFacade.create("myext::JavaBeanExtension");

// setup additional type system
JavaMetaModel jmm =
   new JavaMetaModel("JavaMM", new JavaBeansStrategy());

f.registerMetaModel(jmm);

// use the facade
MyBeanMetaClass jb = MyBeanMetaClass();
jb.setMyProp("test");
f.call("readMyProp", new Object[]{jb}));
```

The called extension file looks like this:

```
import mypackage;

readMyProp(MyBeanMetaClass jb) :
   jb.myProp
;
```

5.9. WorkflowComponent

With the additional support for model transformation, it makes sense to invoke *Xtend* within a workflow. A typical workflow configuration of the *Xtend* component looks like this:

```
<component class="org.eclipse.xtend.XtendComponent">
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
      <metaModelFile value="metamodel1.ecore"/>
   </metamodel>
   <metaModel class="org.eclipse.xtend.typesystem.type.emf.EmfMetaModel">
      <metaModelFile value="metamodel2.ecore"/>
   </metaModel>
   <invoke value="my::example::Trafo::transform(inputSlot)"/>
   <outputSlot value="transformedModel"/>
</component>
```

Note that you can mix and use any kinds of metamodels (not only EMF metamodels).

5.10. Aspect-Oriented Programming in *Xtend* (since 4.2)

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 things without modifying any code, you can make use of around advices that are supported by *Xtend*.

The following advice is weaved around every invocation of an extension whose name starts with 'my::generator::':

```
around my::generator::*(*) :
  log('Invoking ' + ctx.name) -> ctx.proceed()
;
```

Around advices let you change behaviour in an non-invasive way (you do not need to touch the packaged extensions).

5.10.1. Join Point and Point Cut Syntax

Aspect orientaton is basically about weaving code into different points inside the call graph of a software module. Such points are called *join points*. In *Xtend* the join points are the extension invocations (Note that *Xpand* offers a similar feature, see the *Xpand* documentation).

One specifies 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] :
   expression;
```

A point cut consists of a fully qualified name and a list of parameter declarations.

5.10.1.1. Extensions Name

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

```
my::Extension::definition // extensions with the specified name
org::eclipse::xpand2::* //extensions prefixed with 'org::eclipse::xpand2::'
*Operation* // extensions containing the word 'Operation' in it.
* // all extensions
```

> Be careful when using wildcards, because you will get an endless recursion, in case you weave an extension, which is called inside the advice.

5.10.1.2. Parameter Types

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

5.10.1.3. Proceeding

Inside an advice, you might want to call the underlying definition. This can be done using the implicit variable `ctx`, which is of the type xtend::AdviceContext 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(List[Object] params)`. You should be aware, that in advices, no type checking is done.

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

5.10.2. Workflow configuration

To weave the defined advices into the different join points, you need to configure the `XtendComponent` with the qualified names of the Extension files containing the advices.

Example:

```
<component class="org.eclipse.xtend.XtendComponent">
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
      <metaModelFile value="metamodel1.ecore"/>
   </metamodel>
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
      <metaModelFile value="metamodel2.ecore"/>
   </metaModel>

   <invoke value="my::example::Trafo::transform(inputSlot)"/>
      <outputSlot value="transformedModel"/>
   <advices value="my::Advices,my::Advices2"/>
</component>
```

5.10.3. Model-to-Model transformation with *Xtend*

This example uses Eclipse EMF as the basis for model-to-model transformations.

The idea in this example is to transform the data model introduced in the EMF example into itself. This might seem boring, but the example is in fact quite illustrative.

5.10.4. Workflow

By now, you should know the role and structure of workflow files. Therefore, the interesting aspect of the workflow file below is the *`XtendComponent`*.

```
<workflow>
   <property file="workflow.properties"/>
      ...
   <component class="org.eclipse.xtend.XtendComponent">
      <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
         <metaModelPackage value="data.DataPackage"/>
      </metaModel>
      <invoke value="test::Trafo::duplicate(rootElement)"/>
      <outputSlot value="newModel"/>
   </component>
   ...
</workflow>
```

As usual, we have to define the metamodel that should be used, and since we want to transform a data model into a data model, we need to specify only the `data.DataPackage` as the metamodel.

We then specify which function to invoke for the transformation. The statement `test::Trafo::duplicate(rootElement)` means to invoke:

- the `duplicate` function taking the contents of the `rootElement` slot as a parameter
- the function can be found in the `Trafo.ext` file

- and that in turn is in the classpath, in the test.

5.10.5. The transformation

The transformation, as mentioned above, can be found in the `Trafo.ext` file in the `test` package in the `src` folder. Let us walk through the file.

So, first we import the metamodel.

```
import data;
```

The next function is a so-called create extension. Create extensions, as a side effect when called, create an instance of the type given after the `create` keyword. In our case, the `duplicate` function creates an instance of `DataModel`. This newly created object can be referred to in the transformation by `this` (which is why `this` is specified behind the type). Since `this` can be omitted, we do not have to mention it explicitly in the transformation.

The function also takes an instance of `DataModel` as its only parameter. That object is referred to in the transformation as `s`. So, this function sets the name of the newly created `DataModel` to be the name of the original one, and then adds duplicates of all entities of the original one to the new one. To create the duplicates of the entities, the `duplicate()` operation is called for each `Entity`. This is the next function in the transformation.

```
create DataModel this duplicate(DataModel s):
   entity.addAll(s.entity.duplicate()) ->
   setName(s.name);
```

The duplication function for entities is also a create extension. This time, it creates a new `Entity` for each old `Entity` passed in. Again, it copies the name and adds duplicates of the attributes and references to the new one.

```
create Entity this duplicate(Entity old):
   attribute.addAll(old.attribute.duplicate()) ->
   reference.addAll(old.reference.duplicate()) ->
   setName(old.name);
```

The function that copies the attribute is rather straight forward, but ...

```
create Attribute this duplicate(Attribute old):
   setName(old.name) ->
   setType(old.type);
```

... the one for the references is more interesting. Note that a reference, while being owned by some `Entity`, also references another Entity as its target. So, how do you make sure you do not duplicate the target twice? *Xtend* provides explicit support for this kind of situation. *Create extensions are only executed once per tuple of parameters!* So if, for example, the *Entity* behind the target reference had already been duplicated by calling the `duplicate` function with the respective parameter, the next time it will be called *the exact same object will be returned*. This is very useful for graph transformations.

```
create EntityReference this duplicate(EntityReference old):
   setName( old.name ) ->
   setTarget( old.target.duplicate() );
```

For more information about the *Xtend* language please see the *Xtend* reference documentation.

6. *Xpand2*

The *Xpand* language 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**+>.

6.1. 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.

6.2. 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»
«ENDDEFINE»
```

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).

6.3. Statements of the *Xpand* language

6.3.1. 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.*`.

6.3.2. EXTENSION

Metamodels are typically described in a structural way (graphical, or hierarchical, etc.) . A shortcoming of this is that it is difficult to specify additional behaviour (query operations, derived properties, etc.). Also, it is a good idea not to pollute the metamodel 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 metaclasses. You do this by using the *Xtend* language. (See the corresponding reference documentation for details)

An EXTENSION import points to the *Xtend* 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.

6.3.3. 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 metamodel class for which the template is defined.

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

To some extent, templates can be seen as special methods of the metaclass â## 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 metaclasses which inherit from the same superclass, *Xpand* will use the corresponding subclass template, in case the template is called for the superclass. Vice versa, the template of the superclass 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, let us assume you have the following metamodel:

Figure 1.1. Sample metamodel

Assume further, you would have a model which contains a collection of `A`, `B` and `C` instances in the property `listOfAs`. Then, you can write the following template:

```
«DEFINE someOtherDefine FOR SomeMetaClass»
   «EXPAND implClass FOREACH listOfAs»
«ENDDEFINE»

«DEFINE implClass FOR A»
   // this is the code generated for the superclass A
«ENDDEFINE»

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

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

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.

6.3.4. 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»
```

6.3.5. 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.

6.3.5.1. 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»
```

6.3.6. 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. In this case, the specified definition is executed for each element of that collection.

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

6.3.6.1. Specifying a Separator

If a definition is to be expanded FOREACH element of the target expression it is 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.

6.3.7. 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
```

6.3.8. 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»
```

6.3.9. 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 protected region 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»
```

6.3.10. 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 fqn»
   the fully qualified name is: «fqn»;
«ENDLET»
```

6.3.11. 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.

6.3.12. 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»
```

6.3.13. Expression Statement

Expressions support processing of the information provided by the instantiated metamodel. *Xpand* provides powerful expressions for selection, aggregation, and navigation. *Xpand* uses the expressions sublanguage 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 the `toString()` method of `java.lang.Object`). Example:

```
public class «this.name» {
```

All expressions defined by the oArchitectureWare expressions sublanguage are also available in *Xpand*. You can invoke imported extensions. (See the *Expressions* and *Xtend language reference* for more details).

6.3.14. Controlling generation of whitespace

If you want to omit the output of superfluous whitespace you can add a minus sign 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, this mechanism works as follows: 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.

6.4. 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 an non-invasive way (you do not need to touch the generator templates). Because aspects are invasive, a template file containing `AROUND` aspects must be wrapped by configuration (see next section).

6.4.1. 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.

6.4.1.1. Definition Name

The definition name part of a point cut must match the fully qualified name of the join point 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::eclipse::xpand2::* // definitions prefixed with 'org::eclipse::xpand2::'
*Operation* // definitions containing the word 'Operation' in it.
*           // all definitions
```

6.4.1.2. 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 supertype of the corresponding parameter type (the dynamic type at runtime!) of the definition to be called.

Additionally, one can set a wildcard at the end of the parameter list, to specify that there might be an arbitrary number of 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
```

6.4.1.3. 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
```

6.4.2. 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 which provides an operation `proceed()` that 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)`. Please keep in mind that no type checking is done in this context.

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

6.5. 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. A simple generator component configuration could look as follows:

```
<component class="org.eclipse.xpand2.Generator">
   <fileEncoding value="ISO-8859-1"/>
   <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
       <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
   </metaModel>
   <expand value="example::Java::all FOR myModel"/>

   <!-- aop configuration -->
   <advices value='example::Advices1, example::Advices2'/>

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

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

Now, let us go through the different properties one by one.

6.5.1. Main configuration

The first thing to note is that the qualified Java name of the component is `org.eclipse.xpand2.Generator`.

6.5.2. Encoding

For *Xpand*, it is 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.

6.5.3. Metamodel

The property `metaModel` is used to tell the generator engine on which metamodels the *Xpand* templates should be evaluated. One can specify more than one metamodel here. Metamodel implementations are required by the expression framework (see *Expressions*) used by *Xpand2*. In the example above we configured the Ecore metamodel using the *EMFMetaModel* implementation shipped with the core part of the *Xpand* 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. Instead, we specify the name of the property. 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 is 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.

6.5.4. 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="org.eclipse.xpand2.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.

6.5.5. Beautifier

Beautifying the generated code is a good idea. It is very important that generated code looks 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 â## 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.eclipse.xpand2.output.JavaBeautifier"/>
<beautifier
   class="org.eclipse.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 `PostProcessor` Java interface:

```
package org.eclipse.xpand2.output;

public interface PostProcessor {
```

```
    public void beforeWriteAndClose(FileHandle handle);
    public void afterClose(FileHandle handle);
}
```

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

6.5.5.1. JavaBeautifier

The JavaBeautifier is based on the Eclipse Java formatter provides base beautifying for Java files.

6.5.5.2. 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.

6.5.6. 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, .svn
```

If you do not 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"/>
```

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

6.5.7. VetoStrategy

The *Xpand* engine will generate code for each processed FILE statement. This implies that files are written that might not have changed to the previous generator run. Normally it does not matter that files are rewritten. There are at least two good reasons when it is better to avoid rewriting of files:

1. The generated source code will be checked in. In general it is not the recommended way to go to check in generated code, but sometimes you will have to. Especially with CVS there is the problem that rewritten files are recognized as modified, even if they haven't changed. So the problem arises that identical files get checked in again and again (or you revert it manually). When working in teams the problem even becomes worse, since team members will have conflicts when checking in.
2. When it can be predicted that the generator won't produce different content before a file is even about to be created by a FILE statement then this can boost performance. Of course it is not trivial to predict that a specific file won't result in different content before it is even created. This requires information from a prior generator run and evaluation against the current model to process. Usually a diff model would be used as input for the decision.

Case 1) will prevent file writing after a FILE statement has been evaluated, case 2) will prevent creating a file at all.

To achieve this it is possible to add Veto Strategies to the generator, which are implementations of interface `org.eclipse.xpand2.output.VetoStrategy` or `org.eclipse.xpand2.output.VetoStrategy2`. Use `VetoStrategy2` if you implement your own.

`VetoStrategy2` declares two methods:

- `boolean hasVetoBeforeOpen (FileHandle)`

  This method will be called before a file is being opened and generated. Return true to suppress the file creation.

- `boolean hasVeto (FileHandle)`

  This method will be called after a file has been produced and after all configured PostProcessors have been invoked. Return true to suppress writing the file.

Veto Strategies are configured per Outlet. It is possible to add multiple stratgy instances to each Outlet.

```
<component id="generator" class="org.eclipse.xpand2.Generator" skipOnErrors="true">
  <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
  <expand value="templates::Root::Root FOR model"/>
  <fileEncoding value="ISO-8859-1"/>
    <outlet path="src-gen">
       <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier"/>
       <vetoStrategy class="org.eclipse.xpand2.output.NoChangesVetoStrategy"/>
    </outlet>
</component>
```

One `VetoStrategy` is already provided. The `org.eclipse.xpand2.output.NoChangesVetoStrategy` is a simple implementation that will compare the produced output, after it has been postprocessed, with the target file. If the content is identical the strategy vetoes the file writing. This strategy is effective, but has two severe drawbacks:

1. The file has been created at least in memory before. This consumes time and memory. If applying code formatting this usually implies that the file is temporarily written.
2. The existing file must be read into memory. This also costs time and memory.

Much better would be to even prevent the creation of files by having a valid implementation for the `hasVetoBeforeOpen()` method. Providing an implementation that predicts that files do not have to be created requires domain knowledge, thus a standard implementation is not available.

The number of skipped files will be reported by the Generator component like this:

```
2192 INFO  - Generator(generator): generating <...>
3792 INFO  - Skipped writing of 2 files to outlet [default](src-gen)
```

6.6. Example for using Aspect-Oriented Programming in *Xpand*

This example shows how to use aspect-oriented programming techniques in *Xpand* templates. It is applicable to EMF based and *Classic* systems. However, we explain the idea based on the *emfExample* â## hence you should read that before.

6.7. The Problem

There are many circumstances when template-AOP is useful. Here are two examples:

Scenario 1: Assume you have a nice generator that generates certain artifacts. The generator (or cartridge) might be a third party product, delivered in a single JAR file. Still you might want to adapt certain aspects of the generation process â## *without modifying the original generator*.

Scenario 2: You are building a family of generators that can generate variations of the generate code, e.g. Implementations for different embedded platforms. In such a scenario, you need to be able to express those differences (variabilities) sensibly without creating a non-understandable chaos of *if* statements in the templates.

6.8. Example

To illustrate the idea of extending a generator without "touching" it, let us create a new project called `org.eclipse.demo.emf.datamodel.generator-aop`. The idea is that it will "extend" the original `org.eclipse.demo.emf.datamodel.generator` project introduced in the *emfExample*. So this new projects needs to have a project dependency to the former one.

6.8.1. Templates

An AOP system always needs to define a join point model; this is, you have to define, at which locations of a (template) program you can add additional (template) code. In *Xpand*, the join points are simply templates (i.e.

DEFINE .. ENDDEFINE) blocks. An "aspect template" can be declared *AROUND* previously existing templates. If you take a look at the `org.eclipse.demo.emf.datamodel.generator` source folder of the project, you can find the `Root.xpt` template file. Inside, you can find a template called `Impl` that generates the implementation of the JavaBean.

```
«DEFINE Entity FOR data::Entity»
   «FILE baseClassFileName() »
      // generated at «timestamp()»
      public abstract class «baseClassName()» {
         «EXPAND Impl»
      }
   «ENDFILE»
«ENDDEFINE»

«DEFINE Impl FOR data::Entity»
   «EXPAND GettersAndSetters»
«ENDDEFINE»

«DEFINE Impl FOR data::PersistentEntity»
   «EXPAND GettersAndSetters»
    public void save() {

   }
«ENDDEFINE»
```

What we now want to do is as follows: Whenever the *Impl* template is executed, we want to run an additional template that generates additional code (for example, some kind of meta information for frameworks â## the specific code is not important for the example here).

So, in our new project, we define the following template file:

```
«AROUND Impl FOR data::Entity»
   «FOREACH attribute AS a»
      public static final AttrInfo «a.name»Info = new AttrInfo(
         "«a.name»", «a.type».class );
   «ENDFOREACH»
   «targetDef.proceed()»
«ENDAROUND»
```

So, this new template wraps around the existing template called `Impl` It first generates additional code and then forwards the execution to the original template using `targetDef.proceed()`. So, in effect, this is a `BEFORE` advice. Moving the `proceed` statement to the beginning makes it an `AFTER` advice, omitting it, makes it an override.

6.8.2. Workflow File

Let us take a look at the workflow file to run this generator:

```
<workflow>
   <cartridge file="workflow.mwe"/>
   <component adviceTarget="generator"
              id="reflectionAdvice"
              class="org.eclipse.xpand2.GeneratorAdvice">
      <advices value="templates::Advices"/>
   </component>
</workflow>
```

Mainly, what we do here, is to call the original workflow file. It has to be available from the classpath. After this cartridge call, we define an additional workflow component, a so called *advice component*. It specifies *generator* as its *adviceTarget*. That means, that all the properties we define inside this advice component will be added to the component referenced by name in the *adviceTarget* instead. In our case, this is the generator. So, in effect, we add the `<advices value="templates::Advices" />` to the original generator component (without invasively modifying its own definition). This contributes the advice templates to the generator.

6.8.3. Running the new generator

Running the generator produces the following code:

```
public abstract class PersonImplBase {
```

```
   public static final AttrInfo
      nameInfo = new AttrInfo("name", String.class);
   public static final AttrInfo
      name2Info = new AttrInfo("name2", String.class);
   private String name;
   private String name2;

   public void setName(String value) {
      this.name = value;
   }

   public String getName() {
      return this.name;
   }

   public void setName2(String value) {
      this.name2 = value;
   }

   public String getName2() {
      return this.name2;
   }
}
```

6.9. More Aspect Orientation

In general, the syntax for the *AROUND* construct is as follows:

```
<<AROUND fullyQualifiedDefinitionNameWithWildcards
      (Paramlist (*)?) FOR TypeName>>
   do Stuff
<<ENDAROUND>>
```

Here are some examples:

```
<<AROUND *(*) FOR Object>>
```

matches all templates

```
<<AROUND *define(*) FOR Object>>
```

matches all templates with *define* at the end of its name and any number of parameters

```
<<AROUND org::eclipse::xpand2::* FOR Entity>>
```

matches all templates with namespace *org::eclipse::xpand2::* that do not have any parameters and whose type is Entity or a subclass

```
<<AROUND *(String s) FOR Object>>
```

matches all templates that have exactly one `String` parameter

```
<<AROUND *(String s,*) FOR Object>>
```

matches all templates that have at least one `String` parameter

```
<<AROUND my::Template::definition(String s) FOR Entity>>
```

matches exactly this single definition

Inside an `AROUND`, there is the variable `targetDef`, which has the type `xpand2::Definition`. On this variable, you can call `proceed`, and also query a number of other things:

```
<<AROUND my::Template::definition(String s) FOR String>>
   log('invoking '+<<targetDef.name>>+' with '+this)
   <<targetDef.proceed()>>
<<ENDAROUND>>
```

Chapter 2. Built-in types API documentation

1. `Object`

Supertype: none

Table 2.1. Properties

Type	Name	Description
`xpand2::Type`	`metaType`	returns this object's meta type.

Table 2.2. Operations

Return type	Name	Description
`Boolean`	`==(Object)`	
`Boolean`	`<(Object)`	
`String`	`toString()`	returns the String representation of this object. (Calling Java's toString() method)
`Boolean`	`<=(Object)`	
`Boolean`	`!=(Object)`	
`Boolean`	`>(Object)`	
`Integer`	`compareTo(Object)`	Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
`Boolean`	`>=(Object)`	

2. `String`

Supertype: `Object`

Table 2.3. Properties

Type	Name	Description
`Integer`	`length`	the length of this string

Table 2.4. Operations

Return type	Name	Description
`String`	`toLowerCase()`	Converts all of the characters in this String to lower case using the rules of the default locale (from Java)
`String`	`+(Object)`	concatenates two strings
`List`	`toCharList()`	splits this String into a List[String] containing Strings of length 1

Return type	Name	Description
`String`	`toFirstUpper` ()	Converts the first character in this String to upper case using the rules of the default locale (from Java)
`String`	`subString` (`Integer`, `Integer`)	Returns a new string that is a substring of this string.
`String`	`trim` ()	Returns a copy of the string, with leading and trailing whitespace omitted. (from Java 1.4)
`String`	`toFirstLower` ()	Converts the first character in this String to lower case using the rules of the default locale (from Java)
`String`	`toUpperCase` ()	Converts all of the characters in this String to upper case using the rules of the default locale (from Java)
`List`	`split` (`String`)	Splits this string around matches of the given regular expression (from Java 1.4)
`Boolean`	`startsWith` (`String`)	Tests if this string starts with the specified prefix.
`Boolean`	`matches` (`String`)	Tells whether or not this string matches the given regular expression. (from Java 1.4)
`Integer`	`asInteger` ()	Returns an Integer object holding the value of the specified String (from Java 1.5)
`Boolean`	`contains` (`String`)	Tests if this string contains substring.
`Boolean`	`endsWith` (`String`)	Tests if this string ends with the specified prefix.
`String`	`replaceFirst` (`String`, `String`)	Replaces the first substring of this string that matches the given regular expression with the given replacement.
`String`	`replaceAll` (`String`, `String`)	Replaces each substring of this string that matches the given regular expression with the given replacement.

3. `Integer`

Supertype: `Real`

This type does not define any properties.

Table 2.5. Operations

Return type	Name	Description
`List`	`upTo` (`Integer`)	returns a List of Integers starting with the value of the target expression, up to the value of the specified Integer, incremented by one.

Return type	Name	Description
		 </br> e.g. '1.upTo(5)' evaluates to {1,2,3,4,5}
`Boolean`	`>=(Integer)`	
`Boolean`	`==(Integer)`	
`Boolean`	`!=(Integer)`	
`List`	`upTo(Integer, Integer)`	returns a List of Integers starting with the value of the target expression, up to the value of the first paramter, incremented by the second parameter. </br> e.g. '1.upTo(10, 2)' evaluates to {1,3,5,7,9}
`Integer`	`-(Integer)`	
`Integer`	`+(Integer)`	
`Boolean`	`<=(Integer)`	
`Boolean`	`<(Integer)`	
`Integer`	`*(Integer)`	
`Integer`	`-()`	
`Boolean`	`>(Integer)`	
`Integer`	`/(Integer)`	

4. `Boolean`

Supertype: `Object`

This type does not define any properties.

Table 2.6. Operations

Return type	Name	Description
`Boolean`	`!()`	

5. `Real`

Supertype: `Object`

This type does not define any properties.

Table 2.7. Operations

Return type	Name	Description
`Real`	`*(Real)`	
`Boolean`	`>=(Object)`	
`Boolean`	`<=(Object)`	
`Real`	`-()`	
`Boolean`	`==(Object)`	
`Boolean`	`!=(Object)`	
`Boolean`	`<(Object)`	

Return type	Name	Description
`Real`	`-` (`Real`)	
`Real`	`/` (`Real`)	
`Boolean`	`>` (`Object`)	
`Real`	`+` (`Real`)	

6. `Collection`

Supertype: `Object`

Table 2.8. Properties

Type	Name	Description
`Boolean`	`isEmpty`	returns true if this Collection is empty
`Integer`	`size`	returns the size of this Collection

Table 2.9. Operations

Return type	Name	Description
`Boolean`	`contains` (`Object`)	returns true if this collection contains the specified object. otherwise false. returns this Collection.
`List`	`toList` ()	converts this collection to List
`Set`	`toSet` ()	converts this collection to Set
`List`	`flatten` ()	returns a flattened List.
`Set`	`intersect` (`Collection`)	returns a new Set, containing only the elements contained in this and the specified Collection
`String`	`toString` (`String`)	concatenates each contained element (using toString()), separated by the specified String.
`Collection`	`removeAll` (`Object`)	removes all elements contained in the specified collection from this Collection if contained (modifies it!). returns this Collection.
`Collection`	`remove` (`Object`)	removes the specified element from this Collection if contained (modifies it!). returns this Collection.
`Set`	`without` (`Collection`)	returns a new Set, containing all elements from this Collection without the elements from specified Collection
`Collection`	`addAll` (`Collection`)	adds all elements to the Collection (modifies it!). returns this Collection.
`Collection`	`add` (`Object`)	adds an element to the Collection (modifies it!). returns this Collection.

Return type	Name	Description
Set	union (Collection)	returns a new Set, containing all elements from this and the specified Collection
Boolean	containsAll (Collection)	returns true if this collection contains each element contained in the specified collection. otherwise false. returns this Collection.

7. List

Supertype: Collection

This type does not define any properties.

Table 2.10. Operations

Return type	Name	Description
List	withoutFirst ()	
Object	last ()	
Integer	indexOf (Object)	
List	withoutLast ()	
Collection	reverse ()	
Object	first ()	
Object	get (Integer)	

8. Set

Supertype: Collection

This type does not define any properties.

This type does not define any operations.

9. xpand2::Type

Supertype: Object

Table 2.11. Properties

Type	Name	Description
String	name	
Set	allStaticProperties	
String	documentation	
Set	superTypes	
Set	allProperties	
Set	allFeatures	
Set	allOperations	

Table 2.12. Operations

Return type	Name	Description
xpand2::StaticProperty	getStaticProperty (String)	

Return type	Name	Description
xpand2::Feature	getFeature (String, List)	
Boolean	isInstance (Object)	
xpand2::Property	getProperty (String)	
Object	newInstance ()	
Boolean	isAssignableFrom (xpand2::Type)	
xpand2::Operation	getOperation (String, List)	

10. xpand2::Feature

Supertype: Object

Table 2.13. Properties

Type	Name	Description
String	name	
xpand2::Type	returnType	
String	documentation	
xpand2::Type	owner	

This type does not define any operations.

11. xpand2::Property

Supertype: xpand2::Feature

This type does not define any properties.

Table 2.14. Operations

Return type	Name	Description
Void	set (Object, Object)	
Object	get (Object)	

12. xpand2::Operation

Supertype: xpand2::Feature

This type does not define any properties.

Table 2.15. Operations

Return type	Name	Description
List	getParameterTypes ()	
Object	evaluate (Object, List)	

13. xpand2::StaticProperty

Supertype: xpand2::Feature

This type does not define any properties.

Table 2.16. Operations

Return type	Name	Description
`Object`	`get` ()	returns the static value

14. `Void`

Supertype: `Object`

This type does not define any properties.

This type does not define any operations.

15. `xtend::AdviceContext`

Supertype: `Object`

Table 2.17. Properties

Type	Name	Description
`List`	`paramTypes`	
`String`	`name`	
`List`	`paramNames`	
`List`	`paramValues`	

Table 2.18. Operations

Return type	Name	Description
`Object`	`proceed` (`List`)	
`Object`	`proceed` ()	

16. `xpand2::Definition`

Supertype: `Object`

Table 2.19. Properties

Type	Name	Description
`List`	`paramTypes`	
`String`	`name`	
`List`	`paramNames`	
`xpand2::Type`	`targetType`	

Table 2.20. Operations

Return type	Name	Description
`Void`	`proceed` ()	
`String`	`toString` ()	
`Void`	`proceed` (`Object`, `List`)	

17. `xpand2::Iterator`

Supertype: `Object`

Table 2.21. Properties

Type	Name	Description
`Boolean`	`lastIteration`	
`Boolean`	`firstIteration`	
`Integer`	`elements`	
`Integer`	`counter0`	
`Integer`	`counter1`	

This type does not define any operations.

Chapter 3. XSD Tutorial

This tutorial shows how XML and XML Schemas Definitions (XSD) can be used to generate software. It illustrates how XML files are treated as models, XSDs as meta models and how this integrates with oAW. This tutorial is an introduction, for in-depth details see Chapter 4, *XSD Adapter*.

1. Setup

XSD support for oAW comes with oAW 4.3.1 or later. Make sure the following plugins are installed as well:

- XSD - XML Schema Definition Runtime (http://www.eclipse.org/xsd/, available via Ganymede Update Site)
- Web Tools Platform (WTP) (WTP is not required to use oAW XSD support, but helpful, as its provides a nice XML Schema editor and a schema-aware XML editor. (http://www.eclipse.org/webtools/ , available via Ganymede Update Site)

2. Overview

This tutorial explains how you can do code generation with Xtend and Xpand, using XML Schema Definitions as meta models and XML files as models. To keep things easy, the introduced example is a minimalistic one. A text file is generated from contents specified in XML. The general concept of models, meta models and why and when code generation is useful, is not explained. At the end, a deeper view under the hood is taken to understand how XML Schemas are transformed to EMF Ecore models, and which flexibilities/restrictions this approach provides.

All source files listed within this tutorial are also available as an example project wich can be imported into the Eclipse workspace by running *"File" / "New" / "Example..." / "Xpand/ Xtend Examples using an XSD Meta Model" / "M2T custom XML to Text via Xpand (minimal Example)"*. This will create the project `org.eclipse.xpand.examples.xsd.m2t.minimal` project in your workspace. This minimal example is based on "*M2T custom XML to Java via Xpand*" (`org.eclipse.xpand.examples.xsd.m2t.xml2javawizard`) which is more comprehensive and recommended for further reading.

To generate code from XML files with oAW, at least files of the following four types are needed:

- Meta Model (`metamodel.xsd`)
- Model (`model.xml`)
- oAW Xpand Template (`template.xpt`)
- oAW Workflow (`workflow.oaw`)

Figure 3.1. Minimalistic oAW XSD Project

3. Step 1: Create a Project

To create a Project, create an ordinary Xtend/Xpand-Project. This is done in Eclipse by changing to the Xtend/ Xpand perspective and clicking on *"File" / "New" / "Xtend/Xpand Project"*. After entering a name for the project it is created.

After the project is created, support for XSD meta models needs to be activated. Click with your right mouse button on the project and open the properties window. Then go to the "Xpand/Xtend" page, "*enable project specific settings*" and activate the "*XSD Metamodels*" checkbox. There is no need to leave support for any other meta models activated, except you are sure that you want to use one of them, too. Figure 3.2, “Activate XSD Meta Model Support for Project” shows how the configuration is supposed to look like.

Figure 3.2. Activate XSD Meta Model Support for Project

Then, `org.eclipse.xtend.typesystem.xsd` needs to be added to the project's dependencies. To do so open the file `META-INF/MANIFEST.MF` from your project and navigate to the "Dependencies"-tab.

`org.eclipse.xtend.typesystem.xsd` needs to be added to the list of "Required Plug-ins", as it is shown in Figure 3.3, “Required Dependencies for Project” .

Figure 3.3. Required Dependencies for Project

4. Step 2: Define a Meta Model using XML Schema

In case you are not going to use an existing XML Schema Definition, you can create a new a new one like described below. These steps make use of the Eclipse Web Tools Platform (WTP) to have fancy editors.

In Eclipse, click on *"File"*, *"New"*, *"Other..."* and choose *"XML Schema"* from category "XML". Select the project's "src" folder and specify a filename. Clicking on "finish" creates an empty XSD file. It is important that the XSD file is located somewhere within the project's classpath.

This XML Schema consists of two complex data types, which contain some elements and attributes. "complex" in the XSD terminology means that as opposed to simple data types that they can actually have sub-elements and/or attributes. This example is too minimalistic to do anything useful.

The complex Type `Wizard` contains the elements `startpage` , `name` , `welcometext` , and `choicepage` . Except for `choicepage` all elements have to contain strings, whereas the string of `startpage` must be a valid id of any `ChoicePage` . The complex type `ChoicePage` just contains an `id` and a `name` . For oAW it does not make any difference if something is modeled as an XML-attribute or XML-element. Just the datafield's type defines how oAW treats the value.

To get an overview how schemas can be used by the oAW XSD Adapter, see Section 5, “How to declare XML Schemas”

Internally, the oAW XSD Adapter transforms the XSD model to an Ecore model which oAW can use like any other Ecore model. For more information about that, see Section 4, “Behind the scenes: Transforming XSD to Ecore”

Figure 3.4. WTP Schema Editor

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/wizard"
  xmlns:tns="http://www.example.org/wizard"
  elementFormDefault="qualified">

  <complexType name="Wizard">
    <sequence>
      <element name="startpage" type="IDREF" />
      <element name="name" type="string" />
      <element name="welcometext" type="string" />
      <element name="choicepage" type="tns:ChoicePage" />
    </sequence>
  </complexType>

  <complexType name="ChoicePage">
    <sequence>
      <element name="title" type="string" />
    </sequence>
    <attribute name="id" type="ID" />
  </complexType>

  <element name="wizard" type="tns:Wizard" />
</schema>
```

5. Step 3: Create a Model using XML

As the title says, data in XML-Format will be the model. And as a model has to be valid according to a meta model, the XML files must be valid according to the XSD.

In case you are not going to use an existing XML file, you can create a new one like described below. These steps require the Eclipse Web Tools Platform (WTP) to be installed.

In Eclipse, click on *"File", "New", "Other..."* and choose *"XML"* from category *"XML"*. After specifying a filename within folder "src" choose *"create XML file from an XML Schema"* and select you XML Schema Definition file. Telling Eclipse which schema to use has three advantages: Eclipse validates XML files, there is meta model aware code completion while editing and Eclipse creates a xsi:schemaLocation-attribute which tells anyone who reads the XML file where the schema file is located. This tutorial does not use the xsi:schemaLocation-attribute and introduces the schema file in the oAW workflow instead. For all possible ways see Section 5, “How to declare XML Schemas” . It is important that the XML file is located somewhere within the project's classpath.

```
<?xml version="1.0" encoding="UTF-8"?>
<wizard xmlns="http://www.example.org/wizard">
  <startpage>start</startpage>
  <name>My Example Setup</name>
  <welcometext>Welcome to this little demo application.</welcometext>
  <choicepage id="start">
    <title>Wizard Page One</title>
  </choicepage>
</wizard>
```

6. Step 4: Create a Template using Xpand

Create an ordinary oAW Xpand file: Being in the Xpand/Xtend perspective, go to *"File", "New", "xPand template"*. The Xpand language itself is explained by several other oAW documents. Having XSD meta model support activated like described in Section 3, “Step 1: Create a Project” , oAW scans and watches all it's projects for suitable meta models. Based on what is found, the Xpand editor provides meta model aware code completion.

This example imports "`metamodel`" at the beginning, which refers to a file called `metamodel.xsd` that you have created within the project's classpath in Section 4, “Step 2: Define a Meta Model using XML Schema” . The define-block can be understood as a function named "`Root`" which takes one object of type `metamodel::Wizard` as a parameter. This is the meta model's type for the XML's root object. The file-block creates a file named `wizard.txt` and writes the text that is surrounded by the file-block into the file. `name` , `welcometext` and `choicepage.title` are elements or attributes defined in the XSD meta model. Their values are stored within the XML file and this templates inserts them into the generated (`wizard.txt`) file.

```
«IMPORT metamodel»

«DEFINE Root FOR metamodel::Wizard»
«FILE "wizard.txt"»
Name: «name»
Welcometext: «welcometext»
First Page Title: «choicepage.title»
«ENDFILE»
«ENDDEFINE»
```

7. Step 5: Create a Workflow

The workflow ties together model, meta model and templates and defines the process of how to generate code.

To create a new workflow file, switch to the Xpand/Xtend perspective, click on *"File", "New" and "Workflow file"*. After specifying a folder and a filename an empty workflow is created.

The minimalistic approach consists of two steps:

1. Read the Model: This is done by `org.eclipse.xtend.typesystem.xsd.XMLReader` . It needs exactly one `uri` element which defines the XML file. A further nested element of type `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` tells the `XMLReader` which metamodel to use. `XSDMetaModel` can contain multiple `schemaFile` elements. How the schemas are used for the XML file is determined based on the declared namespaces. `modelSlot` defines a location where the model is stored internally, this is like a variable name which becomes important if you want to handle multiple models within the same workflow.
2. Generate Code: This part just does the regular code generation using Xpand and is not specific to the oAW XSD Adapter at all. The generator `org.eclipse.xpand2.Generator` needs to know which meta model to use. This example references the previously declared one. The `expand` element tells the generator to call the definition named `Root` within file `template.xpt` using the contents of slot `model` as parameter. Element `outlet` defines where to store the generates files.

```
<workflow>
  <component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
    <modelSlot value="model" />
    <uri value="model.xml" />
    <metaModel id="mm"
      class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
      <schemaFile value="metamodel.xsd" />
    </metaModel>
  </component>
  <component class="org.eclipse.xpand2.Generator">
    <metaModel idRef="mm" />
    <expand value="template::Root FOR model" />
    <outlet path="src-gen" />
  </component>
</workflow>
```

8. Step 6: Execute Workflow aka Generate Code

Before you actually execute the workflow, or in case of errors, you can use Figure 3.5, “Files of this Tutorial” to double check your files.

Figure 3.5. Files of this Tutorial

To execute the workflow, click with your right mouse button on the workflow file and choose *"Run As", "oAW Workflow"*, as it is shown in Section 8, “Step 6: Execute Workflow aka Generate Code” .

Figure 3.6. Execute Workflow

When executing the workflow, this output is supposed to appear in Eclipse's Console View. If that View does not pop up automatically, you can reach it via *"Window", "Show View", "Console"*.

```
May 25, 2009 3:09:35 PM org.eclipse.emf.mwe.core.WorkflowRunner prepare
INFO: running workflow: /Users/meysholdt/Eclipse/workspace-3.5-M7/org.eclipse.xpand.examples.xsd.m2t.minima
May 25, 2009 3:09:35 PM org.eclipse.emf.mwe.core.WorkflowRunner prepare
INFO:
May 25, 2009 3:09:36 PM org.eclipse.xtend.typesystem.xsd.XSDMetaModel addSchemaFile
INFO: Loading XSDSchema from 'xsd/m2t/minimal/metamodel.xsd'
May 25, 2009 3:09:37 PM org.eclipse.xtend.typesystem.xsd.builder.OawXSDEcoreBuilder initEPackage
INFO: Creating EPackage 'metamodel' from XSDSchema 'file:/.../bin/xsd/m2t/minimal/metamodel.xsd' (http://ww
May 25, 2009 3:09:37 PM org.eclipse.emf.mwe.core.container.CompositeComponent internalInvoke
INFO: XMLReader: Loading XML file xsd/m2t/minimal/model.xml
May 25, 2009 3:09:37 PM org.eclipse.emf.mwe.core.container.CompositeComponent internalInvoke
INFO: Generator: generating 'xsd::m2t::minimal::template::Root FOR model' => src-gen
May 25, 2009 3:09:38 PM org.eclipse.xpand2.Generator invokeInternal2
INFO: Written 1 files to outlet [default](src-gen)
May 25, 2009 3:09:38 PM org.eclipse.emf.mwe.core.WorkflowRunner executeWorkflow
INFO: workflow completed in 657ms!
```

After code generation, there is a file called `wizard.txt` within the `src-gen` folder. Its contents is supposed to look like shown below. You should be able to recognize the structure you've defined within the template file and the contents from your XML model.

```
Name: My Example Setup
Welcometext: Welcome to this little demo application.
First Page Title: Wizard Page One
```

Chapter 4. XSD Adapter

The XSD Adapter allows oAW to read/write XML files as models and to use XML Schemas (XSDs) as meta models. This reference provides in-depth details, for a quick and pragmatic introduction see Chapter 3, *XSD Tutorial* .

1. Prerequisites

Please take a look at Section 1, “Setup”.

2. Overview

The XSD Adapter performs two major tasks:

1. It converts XML Schemas (XSDs) to Ecore models in a transparent manner, so that the Ecore models are hidden from the user. This is done in the workflow as well as in the IDE (to allow XSD-aware code completion for Xtend/Xpand/Check). For details about the mapping see Section 4, “Behind the scenes: Transforming XSD to Ecore” . For details about the workflow integration see Section 3, “Workflow Components”
2. It extends the EmfMetaModel with concepts that are needed for XSDs. Theses are, for example, support for feature maps (needed to handle comments, nested text, CDATA and processing instructions), QNames, EMaps and composed Simpletypes.

3. Workflow Components

The XSD Adapter provides the following workflow components:

3.1. `XSDMetaModel`

The XSDMetaModel loads the specified XSD, transforms them to Ecore models and makes them available for the other oAW components. If XSDs include/import other XSDs or if XML files reference XSDs via schemaLocation, theses XSDs are also loaded (details: Section 5, “How to declare XML Schemas”). The most common scenario is to declare the `XSDMetaModel` within an `XMLReader`:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel id="mm" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
    <schemaFile value="metamodel.xsd" />
    <registerPackagesGlobally value="true" />
  </metaModel>
</component>
```

Another option is to specify an `XSDMetaModel` independently of other components as a bean:

```
<bean id="mymetamodel" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
  <schemaFile value="metamodel.xsd" />
</bean>
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel idRef="mymetamodel" />
</component>
```

Attention: It can lead to errors when XSDs are loaded multiple times, which can only happen when using multiple `XSDMetaModels` within one workflow. The safe way to go is to declare just one `XSDMetaModel` per workflow and reference it from all components that need it.

Properties:

- schemaFile: optional, allowed multiple times: Specifies an XSD file which is being loaded. The path can be a complete URI, or relative to the project root or classpath.
- registerPackagesGlobally: optional, default "false": If true, generated EPackages are registered to `org.eclipse.emf.ecore.EPackage.Registry.INSTANCE`, EMF's global package registry. Warning: when running workflows from your own java code, make sure to remove the generated packages from the registry before the next run!

3.2. XMLReader

The XMLReader reads one XML file which is valid according to the XSDs loaded by the XSDMetaModel. The XML file is loaded as a model and stored in the specified slot. Example:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel idRef="mymetamodel" />
</component>
```

Properties:

- slot: required: The name of the slot which in which the loaded model is stored. Other workflow components can access the model via referring to this slot.
- uri: required: The file name of the XML file which should be read. Absolute URIs, and pathnames relative to the project root or to the classpath are valid.
- metaModel: optional: Specifies the XSDMetaModel (see Section 3.1, " XSDMetaModel ") for the XMLReader. In case no XSDMetaModel is specified, an XSDMetaModel with default configuration is instantiated implicitly. It is important to pay attention that all needed XSDs can be found while the loading process: Section 5, "How to declare XML Schemas".
- useDocumentRoot: optional, default "false": Dealing with XML files as models, most people think of the XML's root element as the model's root object. This is the default used by the XMLReader. But the XML's root element actually has a parent, the so-called DocumentRoot. Additionally the DocumentRoot contains comments/processing instructions and CDATA section which appears before or after the XML's root element, and, most notably, the DocumentRoot contains information about the used namespaces. If useDocumentRoot is set to true, the XMLReader stores the DocumentRoot-Object instead the XML's root element's object to the specified slot.
- option: optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's XMLResource in the loading process. Valid options are documented via JavaDoc in interface org.eclipse.emf.ecore.xmi.XMLResource . Additionally, the XMLReader supports these options:
  - DEFAULT_NAMESPACE: Specifies a default namespace, in case the XML file does not declare one:

```
<option key="DEFAULT_NAMESPACE" val="http://www.dlese.org/Metadata/opml" />
```

  - NAMESPACE_MAP: Specifies a mapping for namespaces, which is applied when loading XML files.

```
<option key="NAMESPACE_MAP">
  <val class="org.eclipse.xtend.typesystem.xsd.lib.MapBean">
    <mapping from="http://www.eclipse.org/modeling/xpand/example/model/wrong"
             to="http://www.eclipse.org/modeling/xpand/example/model/loadcurve" />
  </val>
</option>
```

3.3. XMLWriter

The XMLWriter writes the model stored in a slot to an XML file. If the slot contains a collection of models, each one is written to a separate file. The model(s) must have been instantiated using an XSD-based meta model. Example:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLWriter">
  <metaModel idRef="svgmm" />
  <modelSlot value="svgmodel" />
  <uri value="src-gen/mycurve.svg" />
</component>
```

Properties:

- slot: required: The name of the slot which holds the model or the collection of models which shall be serialized to XML.
- metaModel: required: The instance of XSDMetaModel, which holds the XSD that the supplied models are based on. Also see Section 3.1, " XSDMetaModel "

- uri: required if no uriExpression is specified: The file name of the XML file which should be written. Absolute URIs are valid. Use relative path names on your own risk.
- uriExpression: required if no uri is specified: In the scenario where multiple XML files are written, this provides a mechanism to determine the file name for each of them. The oAW-expression specified in `expression` is evaluated for each file and has to return a file name. The model that is going to be written is accessible in the expression via a variable that has the name specified in `varName`. Example:

```
<uriExpression varName="docroot" expression="'src-gen/'+ecore2xsd::getFileName(docroot)" />
```

- option: optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's XMLResource in the writing process. Valid options are documented via JavaDoc in interface `org.eclipse.emf.ecore.xmi.XMLResource` .

3.4. `XMLBeautifier`

The `XMLBeautifier` uses EMF to load the XML file, formats the mixed content (elements and text contained by the same element) and writes the file back to disk applying a nice indentation for the elements. The `XMLBeautifier` is not intended to be used in combination with the `XMLWriter`, since the `XMLWriter` cares about indentation by itself. Instead, use it for "manually" constructed XML files using Xpand. Since the frameworks for loading/storing XML always load the whole file into a complex data structure in memory, this approach does not scale well for huge XML files. Example:

```
<component class="org.eclipse.xpand2.Generator">
  <metaModel idRef="mm" />
  <expand value="${src-pkg}::${file}::Root FOR '${out}'" />
  <outlet path="${src-gen-dir}" />
  <beautifier class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier">
    <maxLineWidth value="60" />
    <formatComments value="true" />
    <fileExtensions value=".xml, .html" />
  </beautifier>
</component>
```

Properties:

- maxLineWidth: optional: Specifies the number of character after which a linewrap should be performed.
- formatComments: optional, default true: Specifies if formatting should also be applied to comments.
- fileExtensions: optional, default ".xml, .xsl, .xsd, .wsdd, .wsdl": Specifies a filter for which files formatting should be applied. Only files that match one of the specified file extensions are processed.
- loadOption: optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's XMLResource in the loading process. Valid options are documented via JavaDoc in interface `org.eclipse.emf.ecore.xmi.XMLResource` .
- saveOption: optional, can be specified multiple times: Same as loadOption, except for the difference that these options are applied while the writing process. Example:

```
<saveOption key="XML_VERSION" val="1.1" />
<saveOption key="ENCODING" val="ASCII" />
```

4. Behind the scenes: Transforming XSD to Ecore

In the code generation process an XML Schema is transformed to an EMF Ecore model, which is then used as a meta model by EMF. XSD complex data types are mapped to EClasses, XSD simple data types are mapped to EMF data types defined in `org.eclipse.emf.ecore.xml.type.XMLTypePackage` and `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` maps them to oAW data types. The document *XML Schema to Ecore Mapping* explains the mapping's details. http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf

5. How to declare XML Schemas

There are three different ways to declare your XSDs. It does not matter which way you choose, or how you combine them, as long as the XSD Adapter can find all needed schemas.

1. Within the Workflow: `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` can have any amount of `schemaFile` elements.

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="${file}" />
  <metaModel id="mm" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
    <schemaFile value="model/loadcurve.xsd" />
    <schemaFile value="model/device.xsd" />
  </metaModel>
</component>
```

2. Within the XML file: XML files can contain `schemaLocation` attributes which associate the schema's namespace with the schema's filename. If the schema is created using WTP like described in Section 5, “Step 3: Create a Model using XML” , the `schemaLocation` attribute is created automatically.

```
<?xml version="1.0" encoding="UTF-8"?>
<device:Device
  xmlns:device="http://www.eclipse.org/modeling/xpand/example/model/device"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/modeling/xpand/example/model/device device.xsd">
  <device:Name>MyLaptop</device:Name>
</device:Device>
```

3. Within an XSD: If one schema imports another, the `import` element can have a `schemaLocation` attribute, too.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://www.eclipse.org/modeling/xpand/example/model/device"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.eclipse.org/modeling/xpand/example/model/device"
  xmlns:lc="http://www.eclipse.org/modeling/xpand/example/model/loadcurve"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore">

  <import
    namespace="http://www.eclipse.org/modeling/xpand/example/model/loadcurve"
    schemaLocation="loadcurve.xsd">
  </import>

  <complexType name="Device">
    <sequence>
      <element name="Name" type="string" />
      <element name="LoadCurve" type="lc:LoadCurve" />
    </sequence>
  </complexType>

  <element name="Device" type="tns:Device"></element>
</schema>
```