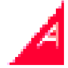
open Generator FrameWork Reference (incomplete)



Autor	Datum	Kommentar
Clemens Kadura	2005-03-15	Translated into English
Jürgen Rühle	2005-05-14	Small fixes

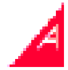
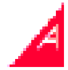


Table Of Content

1 Overview....3
1.1 PropertyProvider....4
2 Frontend....6
2.1 SwingUI....7
2.2 TextUI....7
2.3 AntUI....7
2.4 Outlook....7
3 Meta model....8
3.1 Typ system....8
3.2 Model properties....9
3.3 Identifier and PropertyInvoker....10
3.4 Handling of GenericInstantiator....10
3.5 UML base meta model....12
3.6 Ausblick....12
4 Xpand....13
4.1 Templates....13
4.2 Output....14
4.3 DEFINE....15
4.4 Expression....16
4.5 Condition....18
4.6 Statement....19
4.7 TemplateManager and TemplateReader....23
4.8 ProtectedRegionResolver....23
4.9 SourceWriter....24
4.10 Evaluator....24
4.11 Logging....24
5 Instantiator....25
5.1 GenericInstantiator....25
5.2 Other Instantiators....26



1 Overview

The **open Generator FrameWork** supports a model driven generative development process. The development environment of such a process has in general the following structure:

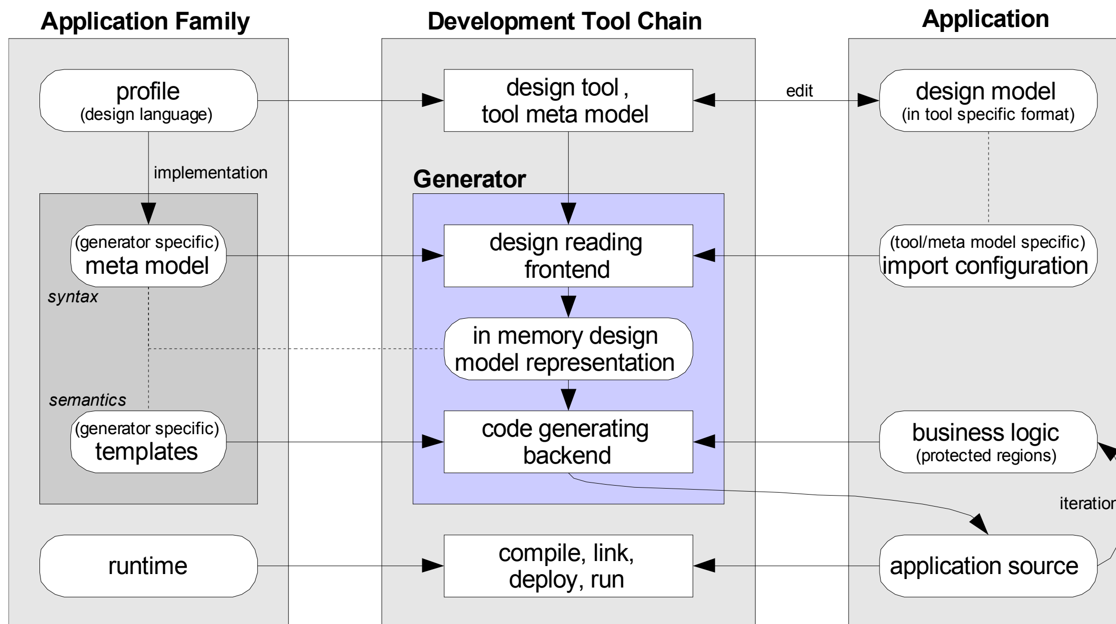


Chart 1.1: generative development environment

The Application is here the desired result of development, whereas the Application Family represents the tool box, that contains the fundamental technical and business components, where the base structure of the application will be assembled or generated from. The characteristic components of a model driven generative Tool Chain are just the design tool and the generator.

The **open Generator FrameWork** is not a monolithic block. The individual modules of the generator can be used seperately or even replaced.

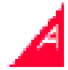
Every modul is typically encapsulated by 2 interfaces:

- the actual module interface
- the Environment of the module, that is the functionality that is required by the module from the embedding application (e.g. access to configuration parameters and logging mechanisms)

The **open Generator FrameWork** is completely implemented in Java and uses only libraries, that are contained in JRE 1.4. How to start Java programs will not further be described here.

This document is intended to be the (currently still incomplete) reference of the modules and its implementations that are contained in version 3.0. For further information please visit the homepage of the project at http://www.architectureware.org, there specially the forum. A useful starting point to examine the

© b+m Informatik AG and Contributors



source code is build.xml and the DTD-sample (what is quite senseless by intention).

1.1 PropertyProvider

The FrameWork uses the `de.bmiag.genfw.PropertyProviderInterface` as a common interface for configuration of the modules.

To support own implementations there is a hierarchy of abstract base classes and one concrete default implementation:

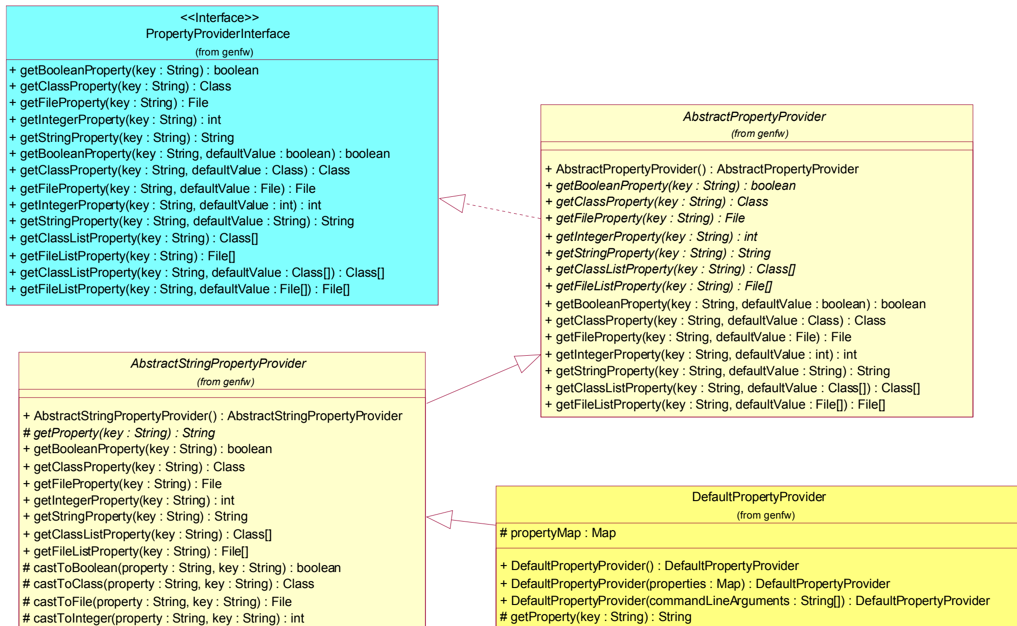
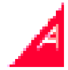


Chart 1.2: PropertyProviderInterface and implementation classes

- `de.bmiag.genfw.AbstractPropertyProvider`
  implements default handling based on the corresponding method without default
- `de.bmiag.genfw.AbstractStringPropertyProvider`
  implements a straightforward string representation for all property types; concrete subclasses only have to implement `String getProperty (String key)` which provides the string representation of property values
- `de.bmiag.genfw.DefaultPropertyProvider`
  is a concrete subclass of `AbstractStringPropertyProvider` implementing `getProperty` in two steps: either the property is provided in the local map or as a system property; the constructor taking a `String[]` interprets each string as *`key=value`* to fill the local map

If the frontend doesn't provide a method to set a specific property, it typically should be enough to set the corresponding system property when starting the VM.

© b+m Informatik AG and Contributors



2 Frontend

The individual modules of the **open Generator FrameWork** can be arbitrarily integrated into the development environment and the development process respectively. The Frontend initialises all necessary modules and binds them into the workflow of the development process. To do this, the Frontend will typically provide a PropertyProvider and Environment implementations.

The currently existing frontend implementations will now be described. They can be used directly or as reference for own implementations.

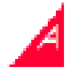
The main class de.bmiag.genfw.Generator is the starting point for the standalone running mode of the generator. When called (e.g. with java -classpath CLASSPATH de.bmiag.genfw.Generator PARAMETER), it processes first all specified command line parameters by instantiating a de.bmiag.genfw.DefaultPropertyProvider (see section). Afterwards the User Interface, that is specified with the parameter de.bmiag.genfw.ui.class is started. This class must implement the interface `de.bmiag.genfw.ui.GeneratorUIInterface`.

The real base of standalone UI implementations is the abstract class `de.bmiag.genfw.ui.AbstractGeneratorUI`. It provides mainly the default implementations of the Environments of the generator modules and other helper methods. Also in case of using an own main class not using GeneratorUIInterface (e.g. when integrated in an IDE) AbstractGeneratorUI can still be reused, since the main class only implements the Java main method and provides the DefaultPropertyProvider.

Here are the properties for `AbstractGeneratorUI` that define the component implementations to be used and the logging properties:

Property	Meaning	default value
`de.bmiag.genfw.instantiator.class`	`de.bmiag.genfw.meta.Instantiator interface` implementation used to instatiate the design	`de.bmiag.genfw.instantiator.xml.GenericInstantiator`
`de.bmiag.genfw.xpand.reader.class`	`de.bmiag.genfw.xpand.TemplateReader interface` implementation used to read the template files	`de.bmiag.genfw.xpand.xpand.TemplateManager`
`de.bmiag.genfw.prresolver.class`	`de.bmiag.genfw.io.ProtectedRegionResolver interface` implementation used to read and write Protected Regions	`de.bmiag.genfw.io.file.SourceReader`
`de.bmiag.genfw.sourcewriter.class`	`de.bmiag.genfw.io.SourceWriterInterface` implementation used to output the generated code	`de.bmiag.genfw.io.file.FileWriter`
`de.bmiag.genfw.logger.level`	global log level, can be modified by specific properties (see below), higher values cause more logging output	no default (only specific loglevel properties have default values)
`de.bmiag.genfw.logger.file`	file to write logged information in	no output into a log file (the UI specific output mechanism is used)
`de.bmiag.genfw.meta.logger.level`	log level for output of the meta model component	3 (alle messages except debug output)
`de.bmiag.genfw.instantiator.logger.level`	log level for output of the instantiator component	3 (alle messages except debug output)
`de.bmiag.genfw.xpand.logger.level`	log level for output of the template reader	3 (alle messages except debug output)
`de.bmiag.genfw.prresolver.logger.level`	log level for output of the protected regions component	3 (alle messages except debug output)
`de.bmiag.genfw.sourcewriter.logger.level`	log level for output of the of the `SourceWriter component`	3 (alle messages except debug output)
`de.bmiag.genfw.evaluator.logger.level`	log level for output of the Evaluator component	3 (alle messages except debug output)
`de.bmiag.genfw.generator.logger.level`	log level for output of the generator itself	3 (alle messages except debug output)

The default extend of supply of the **open Generator FrameWork** contains some special UI implementations that will be further described now.



2.1 SwingUI

The Swing based UI implementation is intended to be a graphical interface to experiment interactively with the generator framework. It is currently not completely realized, specially properties can not be set or changed interactively.

[Screenshot]

[Usage description]

[SwingUI-Properties]

2.2 TextUI

The non interactive TextUI is an example implementation of a frontend. It can also be used as batch frontend directly integrated in a build environment (e.g. Ant). The TextUI realizes the simplest possible generator workflow consisting of the following parts:

- instantiate a meta model object tree from a single input source using the configured instantiator (returns the set of root elements)
- read templates using the configured template reader
- optionally check consistency by expanding the template `Check::Check` for all root elements
- generate output code starting with the selected model elements or root elements respectively by expanding the `Root::Root` template. From these points the generation process can navigate to other model elements and templates depending on the relationships in the actual model and the structure defined in the templates.

The TextUI can be configured using the following properties:

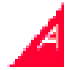
Property	Meaning	default value
`de.bmiag.genfw.textui.blurb`	output copyright, licence and warranty statements to the logger	`true`
`de.bmiag.genfw.textui.check`	activate consistency checks	`true`
`de.bmiag.genfw.textui.select`	the start meta class which model instances shall be used to expand the corresponding templates	no default(if not set de.bmiag.genfw.textui.target is used alternatively)
`de.bmiag.genfw.textui.target`	Wählt das zu generierende Element per vollqualifiziertem Namen ausgehend von den Rootelementen, Namenskomponenten sind durch . getrennt	kein Default (d.h. Generierung der durch `de.bmiag.genfw.textui.select` spezifizierten Elemente bzw. der Rootelemente)
`de.bmiag.genfw.textui.stdout`	activate logging to System.out	`true`

2.3 AntUI

the package `genfwutil` (see project home page or `misc/genfwutil` module in the CVS repository) contains an integration of the generator as an Ant task. This UI can be configured extensively by Plugins.

2.4 Outlook

It is planned for the next release to create a new implementation of the UI and configuration concepts based on a configurable workflow.



3 Meta model

A meta model implementation provides an object instance graph of the design (e.g. an UML model) for the other modules of the generator framework. That's why it is strongly connected with the instantiator module that reads the model data from an external source.

The integration of this component is currently not sufficient. It is planned to change this area significantly in the next versions. So at this point only the user relevant part of the programming model for the delivered base meta model shall be described and its correlation to the standard instantiator. This model will also be supported in future versions beside other implementation alternatives.

The intention of the standard meta programming model is to be able to execute all non trivial manipulations on the instantiated design and keep the template language (and so the templates themselves) as easy as possible. For instance the standard template language Xpand (see section 4) has no support for arithmetic operations on numbers and strings, so numbers and strings must be preprocessed by the meta model implementation and be provided as model properties (as described in section 3.2) to the templates.

3.1 Type system

The base classes of the base meta model are `Element` and `ElementSet`. The following diagram shows the essential methods:

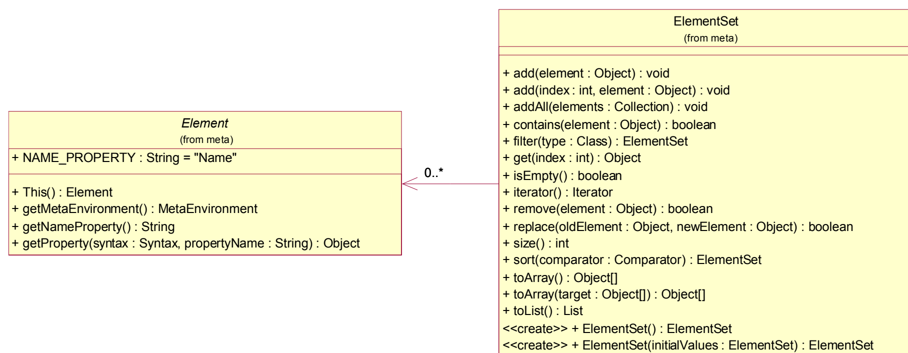
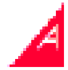


Chart 3.1: Element and ElementSet

`Element` is the base class of all data elements. Collections that shall be processed by the generator must be provided as `ElementSet`. They do implement big amount of the methods of `java.util.Collection`, but not the interface itself, since `add()` has a different semantic: an `ElementSet` can not contain another `ElementSet`. If an `ElementSet` is added to another `ElementSet`, the elements themself will be added as with the `addAll()` method. This behavior was initially intended to simplify the meta model and generator implementation, but there are real doubts currently, if this goal can really be reached (see also Bug #1014422).

Inheritance in the standard meta model is based directly on the Java inheritance model, so only simple inheritance is supported.



Meta model classes (sub classes of `Element`) that shall be instantiable by the `GenericInstantiator` must have a default constructor with `public` access.

There are no restrictions for the naming of meta model classes. But it is important to understand that the templates only refer to the unqualified portion of the name. That's why there should not be multiple classes with the same name in different packages, that are used at the same time. The `GenericInstantiator` returns a warning, if this situation is identified. This is not a problem as long as there are only concrete instances of one of the classes and the other one is a super class of the first one.

3.2 Model properties

The meta model provides the values of a concrete instantiated design model as named model properties to the generator. This can be direct design model data or preprocessed data. Model properties can be of type `String`, `Boolean`, `Element` and `ElementSet`. (actually also other types incl. primitive types should not be a problem, since they are accessed via reflection and the values will be converted if necessary by `toString()`). The standard meta model has two variants to implement model properties:

- methods with public access, which have the same name as the model property (incl. case sensitivity)
- instance variables with public access, which have the same name as the model property (incl. case sensitivity)

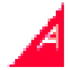
In case that both instance variable and method exists for a property, the method is preferred (actually the instance variable is overridden).

Methods that represent model properties can contain any possible code, specially they can create new meta model element instances dynamically. In this case it is necessary that every new instance is registered in the `de.bmiag.genfw.meta.MetaEnvironment` implementation with its `addElement()` method (at least if it shall be processed by the generator). After an element is registered in the `MetaEnvironment`, the environment is vice versa available via the elements `getMetaEnvironment()` method.

A code sequence to create a new instance dynamically could look like:

```
public MetaClass Property()
{
  element = new MetaClass();
  // register element in aktual MetaEnvironment
  getMetaEnvironment().addElement(element);
  // initialise element
  return element;
}
```

Meta model classes are allowed to define arbitrary additional access and helper methods to be called by other meta model classes during the program flow. These methods don't need to adhere to the property program model. Caching of property values (typically useful for derived properties) is a task



that is currently not provided by default and must be implemented by the meta model.

3.3 Identifier and PropertyInvoker

The standard meta model provides two additional possibilities to implement model properties beyond the meta model classes. If a model property can not be found directly within the meta model class (that means there is neither a `public` method nor a `public` attribute with the name of the property), the following additional steps will be tried:

- Identifier: if the model class has a property with the name `Name` and its value is a subclass of `Element`, the property is searched in the scope of this element (method or variable)
- PropertyInvoker: if an implementation of `de.bmiag.genfw.meta.PropertyInvoker` is registered in the generator's environment, the search for the model property is transfered to this invoker.

In the delivered standard configuration the Identifier class is created by `de.bmiag.genfw.meta.core.ModelElement` (see section 3.5) and the PropertyInvoker will be registered in the `AbstractGeneratorUI` (see section 2). the following configuration properties are used:

Property	Meaning	Default value
`de.bmiag.genfw.baseuml.identifier.class`	Name of the Identifier subclass, that will be used as default Identifier in `createIdentifier()`	`Identifier` (if such a class is available in `CLASSPATH`) else `de.bmiag.genfw.meta.core.Identifier`
`de.bmiag.genfw.propertyNotFoundHandler`	`de.bmiag.genfw.meta.PropertyInvoker` implementation, that will be used to provide global Metamodell properties in the generation process	

3.4 Handling of GenericInstantiator

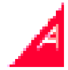
The `GenericInstantiator` is responsible for creating instances of Meta Model classes that represent the design model incl. their subscription in the `MetaEnvironment`.

These Meta Model instances will be initialized with the values of the Design Model using the mappings described in section 5.1.1.

All public fields of the Meta model instances will be filled directly with the value of the corresponding mapping property. If a mapping property is of type `String` a setter method can be used alternatively analogous to Tagged Values.

In most cases (when the mapping is not changed) there is no additional effort necessary because the implementation of the properties is already defined in the included base meta model. Only for design language specific Tagged Values a setter method needs to be implemented in the meta model classes that shall support this property. For example for a Tagged Value named xyZ the corresponding meta class needs a method:

```
public void setXyZ (String value);
```



The first character of the tag name will always be converted to upper case. The implementation can execute arbitrary operations.

Additionally there is a Metamap mechanism that is responsible to define which meta class shall be instantiated for which design model construct. The meta map definitions overwrite the UML-tool specific mappings.

A Metamap is a XML file with the following simple structure:

```
<?xml version="1.0"?>

<!DOCTYPE MetaMap
  PUBLIC "-//b+m Informatik AG//DTD b+m Generator FrameWork MetaMappings 2.1//EN"
  "http://www.bmiag.com/dtds/metamappings_2_1.dtd">

<MetaMap>
  <Mapping>
    <Map>Stereotype or unqualified default design model element name</Map>
    <To>fully qualified meta class name</To>
  </Mapping>
</MetaMap>
```

A Metamap entry can both define a mapping for a specific Stereotype or redefine the relation of a default model element to a specific meta class:

The following entry maps an UML class with stereotype Entity to the Meta class `meta.Entity`

```
<Mapping>
  <Map>Entity</Map>
  <To>meta.Entity</To>
</Mapping>
```

This entry maps all UML attributes without an explicit stereotype to the meta model class `meta.Attribute` (instead of the predefined default mapping to `de.bmiag.genfw.meta.classifier.Attribute`):

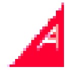
```
<Mapping>
  <Map>Attribute</Map>
  <To>meta.Attribute</To>
</Mapping>
```

There are some restrictions to consider:

The last mapping would also be correct for an explicit stereotype `Attribute`, since the current syntax can not distinguish between these cases. There is currently also no possibility to use the same stereotype for different types of model elements (e.g. `Operation` and `Attribute`) or to use multiple stereotypes on one model element.

There is another interesting feature of the `GenericInstantiator`, the extended stereotype notation. This is intended specially for design tools that don't have sufficient support for tagged values, but it can also be used with all other UML tools. Tagged values can be specified as extension of a stereotype in the following form:

```
Stereotyp, Tag=Value, Tag=Value, ...
```



The separator between stereotype and tags is defined tool specific in the mapping, but it is the comma for almost all provided default mappings. Whitespace around the separator and equality sign resp. is ignored with `trim()`.

3.5 UML base meta model

The base meta model included in the distribution is not intended to be a full blown UML implementation, but it is an easy to use tool box of elements that are differentiated as much as possible for your own application specific profiles. For example only the base model's `Class` supports the Generalization mechanism, whereas in UML this is also available for Associations.

The base meta model does not use the standard Java naming convention, instance variables and methods that represent Model properties have the capitalized name of the property instead.

All derived properties are cached in instance variables with `private` access, that means that changes of the base properties are not taken into account after a derived property is evaluated once.

The design of the base meta model must be read as follows:

- Attribute: base property with instance variable and access method of provided type
- Operation: derived property with access method and cache of provided type
- navigable association without stereotype: base property with instance variable and access method of type of target class(or `ElementSet` if multiplicity > 1)
- navigable association with stereotype `derived`: derived property with access method and cache of type of target class(or `ElementSet`)

[Diagramm Core]

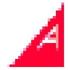
[Diagramm Classifier]

[Diagramm State]

3.6 Outlook

For the next version it is planned to complete the separation of the meta model from the other framework modules. This enables compatability with alternative meta model frameworks. The fate of `ElementSet` will also be decided.

The base meta model will be extended to include UML 2 support (e.g. an UML 2 derived Activity model).



4 Xpand

The actual output generation is handled in the **open Generator FrameWork** by the Xpand engine. Other syntactical representations of templates are possible, but in this documentation only that syntax and semantics are described, that is implemented by the standard `TemplateReader` and `TemplateManager` (see chapter 4.7).

Xpand is a weakly typed language, i.e. the correct types of expressions are only known at runtime and inconsistencies will result in runtime exception messages.

The implementation of the abstract syntax and semantics of Xpand is concentrated in the `de.bmiag.genfw.xpand.syntax` Package. Scanner and Parser for the concrete syntax reside together with the standard `TemplateManager` in the `de.bmiag.genfw.xpand.xpand` Package.

4.1 Templates

Templates are stored in files with the extension `tpl`. A template file consists of any number of `DEFINE` blocks. Within `DEFINE` blocks any text (which does not contain the tag brackets) and comments are allowed. The body of `DEFINE` blocks consists of tags, which are clasped by the characters « and », and text. The tags control the generation whilst the text is written unmodified as output.

4.1.1 Encoding

Xpand is based on the `Character` implementation of the JVM and therefore supports the same pool of characters. For reading the templates any encoding (e.g. `ISO-8859-1` or `UTF-8`) supported by the JVM can be used. Almost all characters required by the standard syntax are part of `ASCII` and should therefore be available in any encoding. The only exception are the tag brackets, for which by default the characters « (unicode `00AB`) and » (unicode `00BB`) are used. Either an encoding containing those characters is used (when it is not the default encoding of your JVM it must be set explicitly!) or the tag brackets must be replaced by other characters (see chapter 4.7).

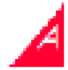
4.1.2 Identifier

Names of properties, templates, namespaces etc. should only contain letters and numbers. The standard `TemplateReader` allows other characters as well by now, but this cannot be guaranteed for future extensions of Xpand.

4.1.3 Comments

Comments are only allowed outside of Tags. Xpand supports both inline comments

```
« REM Text_Comment »
```



and block comments

```
« REM»
  TextComment
« ENDREM »
```

Simple comments must not contain tag brackets. For block comments it is not allowed to contain a `REM` tag, that means block comments are not nestable. For historical reasons it is not allowed for block comments to have a whitespace between the REM keyword and its closing bracket ». Otherwise this white space would be interpreted as a (rather useless) comment (and `ENDREM` results in a parser error).

4.2 Output

The generator output is produced by a `SourceWriter` implementation. All output produced by Xpand statements is transformed to strings (as long as it isn't already) and passed to the `SourceWriter` registered in the `GeneratorEnvironment`. This `SourceWriter` passes the output (after an optional transformation) to the real output channel (typically a file in the file system).

Within Tags whitespace (space, LF and CR) is only treated as separator and will be ignored otherwise. Outside of Tags whitespace is always carried over to the output. To support an appropriate formatting of templates and output (which use typically the same whitespace characters), Xpand supplies two mechanisms to control processing of whitespace.

4.2.1 White space

There are 3 modes of processing `LF` (Unicode `000A`) und `CR` (Unicode `000D`) in text areas within a `DEFINE` block:

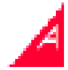
- `VERBATIM`: no transformation occurs, i.e. the characters are copied unmodified from the template into the output
- `NONL`: `LF` and `CR` are ignored completely, formatting of templates with line feeds has no implications to the generated output
- `NORMALIZE`: the combinations `CR LF`, `CR` and `LF` will be interpreted as a single line feed on reading and rendered in the output as the value of the system property `line.separator`

The mode can be set for each block statement (`DEFINE`, `FILE`, `FOREACH`, `IF`, `ELSEIF`, `LET` and `PROTECT`) by explicitly setting one of the key words `VERBATIM`, `NONL` or `NORMALIZE` at the end of the particular start tag. If not set it is carried over from the lexically surrounding block. The default value of `DEFINE` blocks is `NORMALIZE`. The line feed mode is a static property and will be processed only once by the parser. It can not be influenced by the caller of the `DEFINE` block.

A single line feed (`CR LF`, `CR`, or `LF`) can be suppressed by prepending a `NONL` Tag

```
« NONL »
```

© b+m Informatik AG and Contributors



Between the NONL tag and the line feed to be suppressed only white space is allowed (that is also ignored). White space that follows the line feed will not be suppressed.

Line feed can be explicitly created by the NL tag

```
« NL »
```

This renders a normalized line feed (value of system property `line.separator`).

4.2.2 Indentation

Differently from the handling of line feeds indentation is calculated dynamically by the `SourceWriter`. The indentation of lines of a `DEFINE` block can be specified by two parameters (they have to be written before specification of a line feed mode):

- `WS` $\text{Expression}_{\text{Indent}}$: defines the String that will be rendered per indentation level after a line feed
- `INDENT` $\text{Expression}_{\text{Level}}$: defines the level of indentation

These definitions are valid for the whole body of a `DEFINE` block. `WS` and `INDENT` parameter, that are not set explicitly, will be carried over from the calling `DEFINE` block. The default values for `WS` is an empty string and 0 for `INDENT`.

Since indentation can slow down the generation process considerably, `de.bmiag.genfw.io.file.IndentingFileWriter` must be configured explicitly as the `SourceWriter`. `IndentingFileWriter` defines the following semantics for the `INDENT` parameter:

- number: the indent level will be set to this number
- +number: the actual indent level will be incremented by this number.

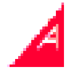
4.3 DEFINE

The central concept of Xpand is the `DEFINE` block. This is the smallest addressable unit. The begin tag consists of a Name (qualified by a namespace), an optional formal parameter list and the Name of that Meta model class, the definition is valid for. The body can contain a sequence of other expansion statements:

```
« DEFINE Identifier_Name [ ( FormalParameterList ) ] FOR Identifier_Metaclass »
  StatementSequence_Body
« ENDDEFINE »
```

The namespace can not be specified explicitly, but emerges from the context. The standard `TemplateReader` uses the name of the template file that contains the `DEFINE` block as namespace.

A `DEFINE` block behaves in the standard meta model regarding resolution of calls like a method in a specified meta class with the name *Namespace*::*Name*. The



resolution of a DEFINE block takes place in context of a concrete instance of the specified meta class (that can be explicitly determined in the standard meta model by invoking This). Overloading of parameter lists is admittedly not supported.

The DEFINE block is the maximal lexical scope of bindings of variables.

4.4 Expression

Expressions support the processing of the information of the instantiated meta model. Xpand provides powerful expressions for selection, aggregation, and navigation. Only rudimentary support for data manipulation is provided. these are better implemented in Java in the meta model.

The value of an expression has one of the types String, Boolean, Element or ElementSet. The value null can actually not be returned as result. In this case almost always a NullEvaluationException is thrown. There are very single cases where null values are allowed (e.g. as target of ISNULL).

Expressions can be bracketed.

4.4.1 Constants

In Xpand the only available constants are String constants. The notation is similar to Java. The following escape sequences are supported:

Escape sequence	represented character
\n	normalised line feed (value of System property line.separator)
\"	"
\\	\
\uxxxx	the Unicode-character specified by the 4 digit code

4.4.2 Invocation

Invocation returns in the simplest case the value of a model property of the actual element (This) or the value of a variable or parameter if the invoked name is bound to the current scope (see LET statement)

```
Invocation ::= Identifier_PropertyName
```

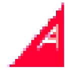
An Invocation can also be a nested expression.

```
Invocation ::= Expression_Target . Identifier_PropertyName
```

At first the specified target expression is processed.

If the target expression evaluates to the type Element, the nested model property is applied to this target element.

If the target expression evaluates to the type ElementSet, the nested property is applied to each element of the target element set. The result will be returned as an ElementSet of all model property values (null values will be ignored).



If the target expression evaluates to another type, an `EvaluationException` is thrown.

An Invocation with an `ElementSet` target is a quite mighty and useful language feature, that allows (specially in combination with the Set arithmetic) complex navigation through the model in a compact written style. So expands for example

```
«EXPAND ImportStatement FOREACH Operation.Parameter.Type SEPARATOR "\n"»
```

(bound on an instance of `Class`) the `DEFINE` block `ImportStatement` (in the current namespace) for each parameter type of each Operation. The set semantics will be maintained, i.e. every type occurs at most one time (see section 4.4.4).

4.4.3 Concatenation

Xpand defines only one String operation, that is Concatenation. A Concatenation evaluates a list of expressions to a String, by using (if necessary) the `Name` property or `toString()`. The syntax is as follows:

```
Expression ::= Expression Expression
```

In some cases it is necessary to put the Concatenation in `()` braces to disambiguate the grammar.

4.4.4 Set Arithmetic

Xpand supports quite powerful set arithmetic to avoid the need to implement a separate model property for each derived set of elements.

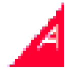
```
Expression ::= Expression + Expression
             | Expression * Expression
             | Expression \ Expression
             | { ExpressionBaseSet AS IdentifierVariable | Condition }
```

The first 3 versions evaluate to union, intersection, and relative complement of values of the specified expressions respectively.

The last case supports the selection of elements from the result set of the base expression, that fulfill the specified condition. To evaluate the condition, the value to be tested is in each case bound to the variable of the specified name. The following expression (bound on an instance of `Class`) evaluates for example all types in the instantiated design, that are neither primitive nor in the package `java.lang`:

```
{ (Operation.Parameter.Type + Operation.ReturnType + Attribute.Type) AS x
  | (x INSTANCEOF Class) && (x.Package.QualifiedName != ":java:lang") }
```

The values of the base expressions must be each of type `ElementSet` (the `DefaultEvaluator` accepts also the result type `Element`, that is converted implicitly to an `ElementSet` with one member).



The precedence of the Set operators is (in order of ascending binding strength): union, intersection, complement.

4.5 Condition

Condition is used in Xpand for conditional statements and selections of sets. Every expression that returns a value of result type `Boolean` is a Condition.

Conditions can also be built with compare operators:

```
Condition ::= Expression == Expression
            | Expression != Expression
```

Compare operations support all data types.

The condition

```
Condition ::= Expression IN Expression
```

tests if the value of the left expression (can be `Element` or `ElementSet`) is contained in the value of the right expression (must be `ElementSet`).

This tests for `null` values or empty sets (Expression must be `Element` or `ElementSet` respectively):

```
Condition ::= ISNULL Expression
            | ISEMPTY Expression
```

This checks meta types dynamically (Expression must evaluate to `Element`)

```
Condition ::= Expression INSTANCOF IdentifierMetaclass
            | Expression HAS_DYNAMIC_TYPE IdentifierMetaclass
```

The first version corresponds to the Java `instanceof`, the second version checks that the value of the expression is exactly of the specified meta class.

Conditions can be combined with the following operators:

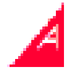
```
Condition ::= Condition || Condition
            | Condition && Condition
            | ! Condition
```

The precedence of the conditional operators is (in order of ascending binding strength): or, and, negation.

4.6 Statement

Statements form the body of a `DEFINE` block and define the generated output. The simplest statement is text that doesn't contain the tag brackets. This is rendered to the output as is.

The following Statements control the generation process.



4.6.1 FILE

```
« FILE ExpressionOutputSpecification [ APPEND ] »
  StatementSequenceBody
« ENDFILE »
```

The `FILE` statement redirects the output generated from it's body statements to the specified target. The expression for the target specification can be a Concatenation. The semantic of the target specification depends on the used `SourceWriter`.

Using a `FileWriter` the target is a file in the file system which name is specified by the expression (relative to the target directory specified by Properties). If `APPEND` is added, it is tried to append the output to the file. Otherwise the file is created new and will be overwritten when it already exists, unless it had already been written during the same generator run.

The body of a `FILE` statement can contain any other statements, especially `FILE` statement can be nested.

If output is eXpanded outside a `FILE` statement, the `FileWriter` simply logs it (as long as it is not only whitespace).

4.6.2 Quote

Every arbitrary character (including control characters, tag brackets or characters that are not available in the current template encoding) can be generated by explicit declaration of it's Unicode.

```
« #xabcd16BitHex »
```

This statement writes a 16-bit (4 hex digits) unicode character to the output.

4.6.3 Invocation

This statement writes the value of a Metamodel Property to the output.

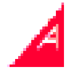
```
« Invocation »
```

Writes the result of the invocation to the putput. If the result is of type `Element`, it's `Name` property will be evaluated (repeating until the result is not of type `Element`). If the resulting value is not of type `String`, `toString()` is called.

4.6.4 EXPAND

Expands another `DEFINE` block (in a separate Variable context), inserts the output at the actual place and continues with the next statement of the actual `DEFINE` block.

```
« EXPAND SUPER [ ( ParameterList ) ]»
```



Expands that DEFINE block, that would be expanded for the super class of the meta class of the actual element (optionally with the specified parameters). If there is no DEFINE block specified for the super meta class (e.g. because it is already assigned to the uppermost meta class Element) an exception is thrown.

```
« EXPAND [ Identifier_Namespace :: ] Identifier_Template [ ( ParameterList ) ] »
« EXPAND [ Identifier_Namespace :: ] Identifier_Template [ ( ParameterList ) ] FOR Expression_Target »
```

Expands the DEFINE block with the specified name for the actual element or the specified target element. The DEFINE block name can have optionally a namespace (what is practically the file name that contains the target DEFINE block) and an optional parameter list. If no namespace is provided, the actual one is used.

An exception will be thrown if the specified target expression can not be evaluated to an existing element or no suitable DEFINE block can be found.

```
« EXPAND [ Identifier_Namespace :: ] Identifier_Template [ ( ParameterList ) ] FOREACH Expression_TargetSet
         [ SEPARATOR Expression ] [ PREFIX Expression ] [ POSTFIX Expression ] »
```

The last version expands the specified DEFINE block for each element of the evaluated target set. While doing so the suitable DEFINE block is determined newly for each element, i.e. polymorphism is fully preserved. The value of the specified SEPARATOR expression is inserted between every two expansions, the values of PREFIX and POSTFIX are inserted before or after evaluation of the complete EXPAND statement, if the specified target set is not empty. An exception will be thrown if the specified target expression can not be evaluated to an ElementSet or no suitable DEFINE block can be found.

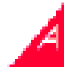
Together with the set arithmetics and white space handling SEPARATOR, PREFIX and POSTFIX allow to effectively control formatting of output.

Before SEPARATOR, PREFIX and POSTFIX the key word USING is allowed. Because of backwards compatibility to version 2.2.1 also USING TRAILER is supported. The expressions for SEPARATOR, PREFIX and POSTFIX can be Concatenations.

4.6.5 FOREACH

```
« FOREACH Expression_TargetSet AS Identifier_Variable EXPAND
          [ SEPARATOR Expression ] [ PREFIX Expression ] [ POSTFIX Expression ] »
  StatementSequence_Body
« ENDFOREACH »
```

This statement expands the body of the FOREACH block for each element of the target set. Different from the EXPAND semantic no new Variable context is created but the actual element is bound to a variable with the specified name in the current context. The meaning of SEPARATOR, PREFIX and POSTFIX is identical to EXPAND...FOREACH. An exception will be thrown if the specified target expression can not be evaluated to an ElementSet.



Within the body of the FOREACH block the bounded variable gets 2 additional properties Index0 and Index1, that hold the 0 and 1 based index of the actual element within the target element set.

The body of a `FOREACH` block can contain any other statements, especially `FOREACH` statement can be nested. The nested index variables get the index properties accordingly.

In reality currently not the element itself is bound to the variable but an instance of `de.bmiag.genfw.meta.Index` that implements the index properties and delegates all other calls to the element. Unfortunately this implementation is not 100% transparent (e.g. not in `==` conditions). Therefore it is sometimes necessary to call the `This` property of the variable that returns the real element (which doesn't provide the index properties).

4.6.6 IF

```
« IF Condition_If » StatementSequence_If
{ « ELSEIF Condition_ElseIf » StatementSequence_ElseIf }
[ « ELSE » StatementSequence_Else ]
« ENDIF »
```

The `IF` statement supports conditional expansion. Any number of `ELSEIF` 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 statements, especially `IF` statement can be nested.

4.6.7 PROTECT

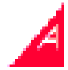
```
« PROTECT CSTART Expression CEND Expression ID Expression [ ENABLED | DISABLED ] »
  StatementSequence_Body
« ENDPROTECT »
```

The `PROTECT` statement encapsulates together with the `ProtectedRegionResolver` (see section 4.8) the whole support of Protected Regions.

The concrete representation of Protected Regions is controlled by the `ProtectedRegionResolver`. The values of CSTART and CEND expressions are used to embrace the Protected Regions marker in the output. They should build valid comment beginning and end strings corresponding to the kind of output.

A Protected Region has an id and an activation state. The id is set by the `ID` expression and must be globally unique (at least for one complete pass of the generator). Activation states can be `ENABLED`, `DISABLED` or `DEFAULT`. The initial value is defined by the `PROTECT` statement, but can be changed by editing the generated code.

During the generation the `ProtectedRegionResolver` reads the current content and state in the code. If there is no state defined in the code, the state defined in the `PROTECT` statement is used. If the resulting state is `DISABLED` or the Protected Region does not exist, the possibly existing actual content of the Protected Re-



gion is dismissed and replaced by the expanded content of the body of the `PROTECT` statement. Otherwise the Protected Region is copied to the output unchanged.

Protected Regions are specially useful, if design changes (e.g. changes to element names) don't have an influence to the id of the protected region.

The standard meta model uses the `Id` property that is filled with the value of the `xmi.uuid` attribute, in case the design tool supports this. It suggests itself to derive the protected region id from this property (e.g. `Id"Imports"`).

If the design tool does not support the `Id` property, Protected Regions can still be used (e.g. using the fully qualified element name as id), but in this case design changes may result in a loss of the content of the protected region.

The state `DISABLED` is useful in cases, when a suitable implementation can be generated, that has a high chance to be changed during the evolution of the architecture, but must be specialized by the developer only in rare cases. In this situation a `DISABLED` protected region can create a hook, what you don't need to take any notice of, but that allows specialization if necessary by changing the state manually to `ENABLED`. In this case of cause changes to the architecture that change the default implementation must be maintained manually.

The expressions for `CSTART`, `CEND` and `ID` can be Concatenations.

4.6.8 LET

```
« LET Expression AS Identifier_Variable »
  StatementSequence_Body
« ENDLET »
```

During the expansion of the body of the `LET` block the value of the expression is bound to the specified variable. The expression will only be evaluated once, independent of the number of usages of the variable within the `LET` block.

4.6.9 ERROR

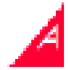
```
« ERROR Expression_Message »
```

The `ERROR` statement aborts the expansion with the specified message. The expression is evaluated to a String and can be a Concatenation.

4.7 TemplateManager and TemplateReader

A `TemplateReader` implementation reads Template definitions from a specified source and provides them to the framework as a `TemplateManager`. Especially it implements the evaluation of (qualified) template names (e.g. out of an `EXPAND` statement) to a concrete `DEFINE` block.

The standard `TemplateReader` `de.bmiag.genfw.xpand.xpand.TemplateManager` can be configured by the following properties:



Property	Meaning	default value
`de.bmiag.genfw.xpand.path`	One or many (separated by `File.pathSeparator`) directories, that contain the Templates to be read	No default
`de.bmiag.genfw.xpand.codepage`	The encoding of the Template files	The platform default encoding (on some platforms `US-ASCII` is the default encoding, that is not not usefull for templates, therefore it should be set explicitly)
`de.bmiag.genfw.xpand.parser.brackets`	Must consist of exactly two different characters that are used as tag brackets. For instance de.bmiag.genfw.xpand.parser.brackets=<> would embrace the tags with < and >, « and » would be normal text. To render the characters < and >, Unicode escapes must be used.	«» (Unicode `00AB` and `00BB`)
`de.bmiag.genfw.xpand.parser.debug`	Activiates the debug output of the Parser . This makes only sense to debug the parser implementation and should not be used for a normal run.	`false`

Usually it is not necessary to create an application specific implementation. The `genfwutil` package contains some useful implementations.

4.8 ProtectedRegionResolver

The `ProtectedRegionResolver` implementation defines the formatting of Protected Regions and the assignment of Id to concrete content.

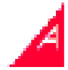
The standard implementation is `de.bmiag.genfw.io.file.SourceReader`. It can be configured by the following properties:

Property	Meaning	default value
`de.bmiag.genfw.sourcereader.path`	one or many (separated by `File.pathSeparator`) directories, that contain the Protected Regions to be read.	no Protected Regions, this property doesn't need to be specified, if `PROTECT` is not used.
`de.bmiag.genfw.sourcereader.codepage`	the encoding of the files that contain Protected Regions	the platform default encoding
`de.bmiag.genfw.sourcereader.excludes`	a list of patterns (separated by whitespace) with at most one * at the begining and end), that define files to be ignored (e.g. editor backups). The default exclude list is not affected by this property (see de.bmiag.genfw.sourcereader.defaultexcludes)	no additional excludes
`de.bmiag.genfw.sourcereader.defaultexcludes`	specifies if the list of default excludes shall be used (in addition to the excludes defined in `de.bmiag.genfw.sourcereader.excludes`). The default exclude list contains the helper files that are typically used by an IDE: 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	`true`
`de.bmiag.genfw.sourcereader.checkpath`	specifies if the value of `de.bmiag.genfw.sourcereader.path` is checked against the property `de.bmiag.genfw.filewriter.path`, to avoid overriding of Protected Regions.	`true` (should only be changed with good reason since loss of data could occur otherwise)
`de.bmiag.genfw.sourcereader.base64`	specifies if the Protected Region ID shall be encoded by Base64. This allows the use of any characters in the ID (inclusive characters that are relevant for the syntax of Protected Regions or the target language respectively). This setting only influences the output of Protected Regions. `SourceReader` can read and distinguish both variants independent of the value of the property.	`false` (may be subject to change in a future release)
`de.bmiag.genfw.sourcereader.report`	specifies if ID and source file of protected regions that were not used during the generation process should be logged. If additionally `de.bmiag.genfw.sourcereader.dumppath` is set, all protected regions that were not used, will be written in an own file named with the Base64 encoded ID.	`false`
`de.bmiag.genfw.sourcereader.dumppath`	path where all unused Protected Regions shall be written	no dump

In some cases an application specific implementation can be useful. For instance if the exported UUIDs (`Id` attribute of `ModelElement` in the base meta model) are different after a version change of the design tool, a special adapter can be added before `SourceReader`, that maps new Ids to the old one's to be used for a migration run.

4.9 SourceWriter

The `SourceWriter` implementation processes the generated target output. The standard implementation `de.bmiag.genfw.file.io.FileWriter` writes it unmodified into files. `de.bmiag.genfw.file.io.IndentingFileWriter` processes indentation as described in section 4.2.2. Both use the following configuration properties:



Property	Meaning	default value
`de.bmiag.genfw.filewriter.path`	the root directory where to put generated files	no default
`de.bmiag.genfw.filewriter.codepage`	the Encoding to be used to write generated output.	platform default encoding

4.10Evaluator

The Evaluator Interface is the central extension interface of Xpand. Every generation, evaluation and/or model property access is routed indirectly through a chain of evaluator implementations. The standard implementation `de.bmiag.genfw.xpand.DefaultEvaluator` implements the documented behavior, i.e. forwarding to the real implementation.

<<Interface>>
EvaluatorInterface
(from xpand)

+ init(env : EvaluatorEnvironment) : void
+ generate(definition : Definition, element : Element, env : GeneratorEnvironment) : void
+ generate(statement : Statement, element : Element, env : GeneratorEnvironment) : void
+ eval(expression : Expression, element : Element, env : GeneratorEnvironment) : Object
+ evalToElement(expression : Expression, element : Element, env : GeneratorEnvironment) : Element
+ evalToElementSet(expression : Expression, element : Element, env : GeneratorEnvironment) : ElementSet
+ evalToString(expression : Expression, element : Element, env : GeneratorEnvironment) : String
+ evalToBoolean(expression : Expression, element : Element, env : GeneratorEnvironment) : boolean
+ isTrue(condition : Condition, element : Element, env : GeneratorEnvironment) : boolean
+ getProperty(invocation : Invocation, parameter : Object[], element : Element, env : GeneratorEnvironment) : Object

chart 4.1: EvaluatorInterface

Implementations could contain tracing or debugging functionality or extend the Xpand meta model interface (similar to the Invoker mechanism).

The following properties are used to configure the evaluator chain used by `AbstractGeneratorUI` (see Chapter 2):

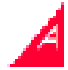
Property	Meaning	default value
`de.bmiag.genfw.xpand.evaluator`	List of classes that implement `de.bmiag.genfw.xpand.EvaluatorInterface`. They are used during the generation in the specified order as Evaluators	no additional Evaluators
`de.bmiag.genfw.xpand.appendDefaultEvaluator`	specifies if `de.bmiag.genfw.xpand.DefaultEvaluator` shall be appended automatically at the end of the evaluator chain	`true`

4.11Logging

With `de.bmiag.genfw.xpand.logger.level=4` additional log output can be activiated (e.g. which template files were read and which definitions were contained). This is helpful to find errors. (E.g. if template files were found, but non of the definitions, a codepage problem could exist).

The following properties control the output of special traces (requires log level ≥ 3), to be used to reconstruct generation:

Property	Meaning	default value
`de.bmiag.genfw.xpand.logger.define`	activates `DEFINE trace`	`false`
`de.bmiag.genfw.xpand.logger.file`	activates `FILE trace`	`true`



5 Instantiator

An Instantiator reads a model in several kinds of representation (e.g. a XMI file) and provides it as instantiated meta model to the other modules of the generator framework.

Since the encapsulation of the meta model component is still work in progress, a detailed documentation can not provided at this point in time.

5.1 GenericInstantiator

All relevant information for `GenericInstantiator` regarding meta model programming model were already provided in chapter 3.4.

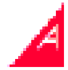
The following properties are used to configure the `GenericInstantiator`:

Property	Meaning	default value
`de.bmiag.genfw.instantiator.design`	Name of the file that contains the design `de.bmiag.genfw.instantiator.xml.io.StreamFactory` derives from this heuristically the XML `InputSource`	no default
`de.bmiag.genfw.instantiator.xmlmap`	Name of the mapping file that matches to the used design tool this is searched first in the `CLASSPATH` and afterwards in the file system and will be evaluated against `xml2meta.dtd`	no default
`de.bmiag.genfw.instantiator.metamap`	Name of the application specific stereotype mapping file This is searched only in the file system and will be evaluated against `metamap.dtd`	no default
`de.bmiag.genfw.instantiator.tooladapter.class`	Name of the class that handles the design tool specific post processing of the instantiated meta model	no default

5.1.1 Tool support

The support of a specific combination of design tool and base meta model consists of a XML mapping and a tool adapter. For the simplified UML base meta model they are already available for the following design tools (at 2005-02-21):

Tool	Version	xmlmap prefix `de/bmiag/genfw/instantiator/toolsupport/baseuml/`	tooladapter.class package prefix `de.bmiag.genfw.instantiator.toolsupport.baseuml`	remarks for further details see mapping file or homepage
ARIS UML-Designer	2.0?	`aris/aris_xmi11_all.xml`	`aris.ArisAdapter`	not tested yet, probably incomplete, compatibility with other versions unknown
Artisan	4.4.2?	`artisan/artisan_xmi13_all.xml`	`artisan.ArtisanAdapter`	not tested yet, probably incomplete, compatibility with other versions unknown
Enterprise Architect	2.5? - ?	`ea/ea25_xmi11_all.xml`	`ea.EAAdapter`	incomplete, specially no support for stereotypes and Tagged Values (and others) for Operation and Attribute because of missing xmi.id in the export
	2.5? - ?	`ea/ea25_xmi11_fixed_all.xml`	`ea.EAAdapter`	supports Stereotypes and Tagged Values for all model elements, requires post processing of the XMI with FIXEAXMI (to be questioned)
	4.1 - ?			to be questioned (not compatible with existing base meta model because of UML 2.0-Activity-Support)
Innovator Object	8.1.02 - ?	`innovator/innovator_xmi11_all.xml`	`innovator.InnovatorAdapter`	minor limitations in productive use
Metamill	3.1 - ?	`metamill/metamill31_xmi12_all.xml`	`metamill.MetamillAdapter`	incomplete quick hack (but in productive use)
Magic Draw	6.? - ?	`magicdraw/magicdraw_xmi11_all.xml`	`magicdraw.MagicDrawAdapter`	not tested yet, but nearly complete, not suitable for newer versions
	9?	`magicdraw/magicdraw_xmi12_v9_all.xml`	`magicdraw.MagicDrawAdapter12`	?
Poseidon	1.6	`poseidon/poseidon16_xmi12_all.xml`	`poseidon.PoseidonAdapter`	?
	2.x - 3.0	`poseidon/poseidon20_xmi12_all.xml`	`poseidon.PoseidonAdapter`	in productive use
Rose + UniSys XML Tools	from 2000e Exporter 1.3.2	`rose/rose_unisys132_xmi10_all.xml`	`rose.RoseAdapter`	recommended Rose-Unisys-XMI-combination, in productive use, only minor shortcomings, newer UniSys-versions give worse results
Together	6.1	`together/together61_uml13_xmi12_cls.xml`	`together.TogetherAdapter`	incomplete, currently only class diagram support
XDE	1.5?	`xde/xde_xmi11_all.xml`	`xde.XDEAdapter`	in productive use, but significant problem in export



There are variants for some mappings that end with `cls.xml` or `sta.xml` instead of `all.xml`. They support in each case only a part of the meta model (namely class diagrams or state and activity diagrams), what could possibly speed up the instantiation.

It is easily possible to support other design tools or other versions respectively (assuming standard conform XMI support to some extent). A detailed documentation shall not be given at this point in time, since the mapping syntax is unnecessarily ugly. Use support of the project maintainers, forum or source code instead.

The tool adapter can be used if needed as application specific hook for model transformations, but it is recommended to implement this by a specialisation of the UI (as for instance AntUI).

5.2 Other Instantiators

The `genfwutil` package contains a series of useful Instantiator implementations, amongst them a `CompositeInstantiator`, to build a total model out of several single components, a `XMLInstantiator`, to map XML files into the meta model and the `VisioInstantiator` for reading of Visio diagrams.