



ingo herwig <ingo@wemove.com>
Giuseppe Platania <gplatani@de.ibm.com>

WCMF GENERATOR

This document describes the use of the wCMFGenerator version 2.1 within the wCMF application development.

SYSTEM REQUIREMENTS

- wCMFGenerator Files wcmf_generator_2.1.zip
- Java 1.5
- UML modeling tool with XML export capability

TABLE OF CONTENTS

1	INTRODUCTION.....	1
2	INSTALLATION OF THE GENERATOR.....	2
3	FUNDAMENTALS.....	2
3.1	GENERATOR OVERVIEW.....	2
3.2	WCMF UML PROFILE.....	3
3.2.1	DEFINITION OF THE DATA MODEL.....	3
3.2.2	STEREOTYPES FOR MODELLING THE DATA.....	4
3.2.3	INHERITANCE.....	7
3.2.4	ASSOCIATIONS.....	8
3.2.4.1	Association types.....	8
3.2.4.2	Multiple associations.....	9
3.2.5	DEFINITION OF THE USER INTERACTION.....	9
3.2.6	STEREOTYPES FOR MODELING THE USER INTERACTION.....	10
3.2.7	STEREOTYPES FOR MODELING THE APPLICATION CONFIGURATION.....	12
4	USING THE GENERATOR.....	13
4.1	MODELING THE APPLICATION.....	13
4.2	CONFIGURATION OF THE GENERATOR.....	18
4.3	RUNNING THE GENERATOR.....	19
4.4	GENERATION RESULT.....	20
5	MODIFICATION OF THE GENERATOR.....	25

1 INTRODUCTION

The wCMFGenerator provides a tool, which suggests a methodical procedure for the wCMF application development.

The goal is to describe and maintain all essential aspects of the application in a separate, graphical model (build with UML), and to generate the code from it (MDA approach). The advantage of this method is, that the condensed form of the graphical model makes the understanding of the project's subject easier and that the different project artifacts (classes, configuration files) can be generated automatically and therefore kept consistent. The generator supports the modeling of the data model, user interaction and the application's configuration.

After the automated code generation the developer can fully concentrate on the implementation of the business logic – the specific behavior of the application's controllers and the views for user interaction. As a matter of course the model can be adapted even after inserting custom code into the generated files. Code, which was created manually, is clearly separated from automatically created one via protected areas.

The wCMFGenerator is based on the Generator Framework *openArchitectureWare*

(<http://www.openarchitectureware.org>). An overview of modeling tools can for instance be found under http://www.objectsbydesign.com/tools/umltools_byProduct.html or <http://www.jeckle.de/umltools.htm>. Requirements for the tool are XML export and support of stereotypes and tagged values. Currently Enterprise Architect¹ is directly supported, other tools have to be evaluated.

2 INSTALLATION OF THE GENERATOR

A Java installation presumed, the archive `wcmf_generator_2.1.zip` must be extracted to any directory.

The wCMFGenerator has the following directory structure:

```
lib/ generator libraries
mappings/ mapping of model stereotypes to classes of the generator's meta model
metamodel/ wCMF meta model in Enterprise Architect format, wCMF UML profile
model/ wCMF template model
templates/
    with_domain_classes/ generator templates for the creation of an application with
                        domain classes
wcmf/ the framework including the default application template
default_ea.properties project settings for the wCMF template model
license.txt license text for the generator
readme.txt note on how to start
run.bat batch file for starting the generator
wCMFGenerator.jar generator application
```

3 FUNDAMENTALS

3.1 GENERATOR OVERVIEW

The generator creates the necessary files for a wCMF application from an UML model (Figure 1). This is achieved by traversing over the individual parts of the model's representation, which was created in the memory beforehand. How the model's individual parts are incorporated in the generated files is defined by a template language. When the generator, while it's traversing the model, encounters a model element, for which a rule is defined in the template files, this rule will be executed and the corresponding part written to the file. More information on how the generator works can be found under <http://www.openarchitectureware.org>.

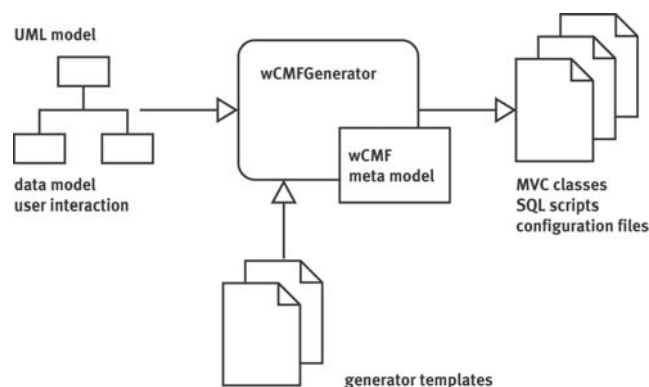


Figure 1 Functioning of the generator

The UML model must be provided as XMI file. This format is supported by a large variety of modeling tools. Because the XML formats can differ from tool to tool, the import of the generator can be customized.

¹ See <http://www.sparxsystems.com.au/>

3.2 WCMF UML PROFILE

An important presupposition for modeling a wCMF application is the knowledge of the wCMF meta model, which formally describes the provided elements. The wCMF meta model is defined as an UML profile, which consists of stereotypes and tagged values. These elements are wCMF specific extensions of the UML meta model. The wCMFGenerator recognizes these elements and creates the corresponding Code for them².

There are three areas in which a wCMF application is modeled:

1. *Definition of the data model:* Configuration of the persistence layer, i.e. which entity types are known to the application and which attributes they have (e.g. „article with the attributes headline and text“).
2. *Definition of the user interaction:* configuration for the presentational layer, i.e. which GUI views are needed and which actions can be performed by the application's users (e.g. „in the login screen the login action can be performed“).
3. *Configuration of the application:* configuration of the wCMF application, i.e. The database connection.

The following sections first give an introduction in how to model the data and then introduce the stereotypes, which are used.

DEFINITION OF THE DATA MODEL

A wCMF application is generally used to maintain any types of content. The different types of content (*articles, images* etc.) are defined in the so called data model. For a better understanding we first look at the persistence layer of the wCMF, because this layer implements the application's data model.

The persistence layer of the wCMF basically consists of the `PersistentMapper`³ classes, which load, store, delete and create instances of the `PersistentObject` class. The `PersistentObjects` hold the content, which is maintained by the application.

To make the implementation of hierarchical organized content easier, a class `Node` exists in the wCMF, which inherits from `PersistentObject`. `Node` instances can contain other `Node` instances (*children*) and can be contained in other `Node` instances, respectively (*parent*). In this way trees can be defined. The content attributes of the `Node` class (called *values* in the wCMF) are not predefined. They are created during runtime by the method `setValue(name, value)`. Therefore `Node` is considered as the universal entity class. Different entity definitions are distinguished via the `type` attribute.

The `PersistentMapper` classes connect the data storage (database, XML files) with the entity objects. They contain the entity definitions, i.e. the type, it's attributes and the relations between different entity types.

It has proved as useful to define one mapper class for each entity type and each data storage. For instance an `ArticleRDBMapper` would store `Node` objects of the type `Article` in a database table named `article`. When using the base class `NodeUnifiedRDBMapper` the only programming task would be the implementation of some methods describing the entity type. After this `Article` instances would be available in the application.

As a summary the implementation of the data model when using a relational database (which would commonly be the case) follows these 4 steps:

1. Creation of a database scheme with an individual table for each entity type.
2. Creation of subclasses of the `NodeUnifiedRDBMapper` class, which implement the methods demanded from the abstract base classes.
3. Declaration of the entity types in the configuration file.

² Note, that the generator will probably not create useful output for UML elements which are not part of the wCMF meta model.

³ A detailed documentation of the wCMF classes comes with the framework.

4. Optionally implementation of a subclass of `Node` for each entity type.

Of course it's possible to do all this manually. Using the `wCMFGenerator` these tasks will be carried out automatically. To accomplish this the generator needs an UML model of the entity types, which must be created with specific stereotypes.

STEREOTYPES FOR MODELLING THE DATA

The generator knows the following stereotypes for data modeling:

Name	UML meta class	Meaning	Example
WCMFNode	Class	Entity type WCMFNodes must inherit from the framework class <code>Node</code> .	Article
WCMFValue	Attribute	A <code>Node</code> value type used in wCMF. Those values are persistent.	an Article's headline
WCMFValueRef	Attribute	ReadOnly-Reference to an attribute of another entity type ⁴ . Advantage: The referenced content type doesn't have to be loaded completely. ValueRefs are not persistent.	If the node Author has Article nodes as children, the Author's name can be referenced in the Article node.
WCMFManyToMany	Class	It is used to realize a many to many relation between 2 WCMFNodes. It must inherit from the framework class <code>Node</code> .	To link many Authors with many Articles.
WCMFRelationEnd	AssociationEnd	An association end used in WCMF used to define a foreign key name.	The name of a foreign key column.

For each stereotype tagged values are defined, which are displayed in the following table.

Stereotype	Tagged value	Meaning	Values	Default value
WCMFNode / WCMFManyToMany	initparams	Name of the configuration file's (config.ini) section, in which the initial parameters for the corresponding mapper are defined		database
	is_ordered	Declares, if the content type can be sorted ⁵ . This is deprecated in 2.1. use <code>orderby</code> instead.	true, false	true
	is_soap	Define if the type should be exposed to the SOAP interface.	true false	true
	orderby	Definition of default sorting. Possible values: 'none' (no order), 'sortkey' (generates a 'sortkey' column, that is used for explicit sorting) or any the name of any WCMFValue defined in the node optionally followed by [ASC DESC] e.g. 'name ASC'	None sortkey anyvalue	none

- 4 The referenced entity type must be directly related to the type holding the reference, i.e. it must be a child or parent node. If the relation isn't clear, unexpected behaviour could be the result (for instance, if `Image` instances are referenced by the name attribute of an `Article` and one `Article` is allowed to contain more than one `Image`, the reference holds the name of the first `Image`).
- 5 An attribute `sortkey` is added, if the content type can be sorted.

Stereotype	Tagged value	Meaning	Values	Default value
		default="none"		
	is_searchable	Indicates whether this type should be included in the default search.	true false	true
	table_name	The name of the database table. If not given the name of the class will be taken.		class name
	pk_name	The name of the primary key column. The generator will add this automatically if there is no appropriate attribute.	a single value or ' ' separated list of values (e.g. fk_user_id fk_role_id)	id
	child_order	The order of the associated children e.g. for Recipe: Image Info AdminInfo.	a single value or ' ' separated list of values	
	parent_order	The order of the associated parents.	a single value or ' ' separated list of values	
	display_value	The value that is displayed in a list.	any WCMFValue in this type.	
WCMFValue	app_data_type	The attribute's application datatype ⁶ . This can be used in the application to group attributes and execute special logic on them.	DATATYPE_DONTCARE, DATATYPE_ATTRIBUTE, DATATYPE_ELEMENT, DATATYPE_IGNORE	DATATYPE_ATTRIBUTE
	db_data_type	The attribute's database type. This will be used in the table definition.	according to the database e.g. INT, VARCHAR, TEXT, ...	VARCHAR(255)
	is_editable	Declares, if the attribute is editable.	true false	true
	input_type	Definition of the attribute's input control in the HTML form ⁷ .	text, password, textarea, select, radio, checkbox, file, fileex, fckeditor, filebrowser, linkbrowser + additional definitions	text
	display_type	The HTML display type for the attribute e.g. image ⁸ .	image, text	text
	restrictions_match	Regular expression, which must be matched by the attribute value ⁹ .	e.g. '[0-3][0-9]\.[0-1][0-9]\.[0-9][0-9][0-9][0-9]' for date values	
	restrictions_not_match	Regular expression, which must not be matched by the attribute value ¹⁰ .	See above	
	restrictions_description	A text describing the restrictions, which will be shown in case of an error.		
	column_name	The name of the database column. If not given the attribute name will be used.		attribute name

6 The application's data types are globally defined e.g. In PersistentObject.

7 The interpretation of the input_type is done by DefaultControlRenderer or its subclasses.

8 The interpretation of the display_type is done by DefaultValueRenderer or its subclasses.

9 The PHP function preg_match is used for comparison.

10 The PHP function preg_match is used for comparison.

Stereotype	Tagged value	Meaning	Values	Default value
WCMFRelationEnd	fk_name	The name of the foreign key column. The generator will add this if there is no appropriate attribute. If not given the generator will add 'fk_parent_id'		fk_[parent]_id
WCMFValueRef	reference_type	Content type, whose attribute is referenced		
	reference_value	The content type's referenced attribute		

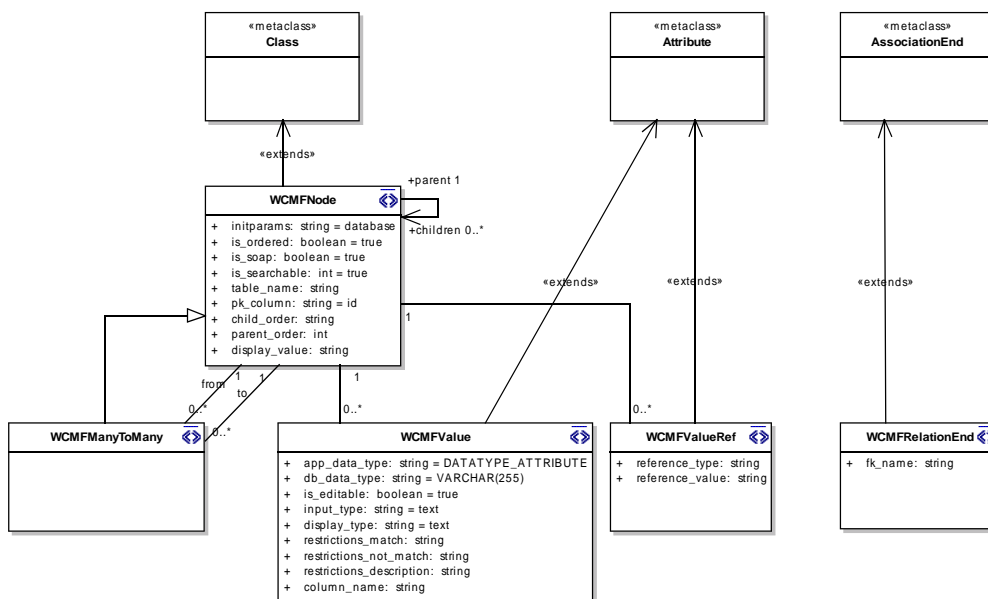


Figure 2 Part of the wCMF UML Profile (data modeling)

Figure 2 illustrates the part of the wCMF UML profile, which is relevant for the modeling of the entity types. As the figure shows, the stereotype WCMFNode and WCMFManyToMany are used for classes while the stereotypes WCMFValue and WCMFValueRef are used for attributes. WCMFNodes can contain other WCMFNodes, their attributes are WCMFValues or WCMFValueRefs, of which they can contain as many as required.

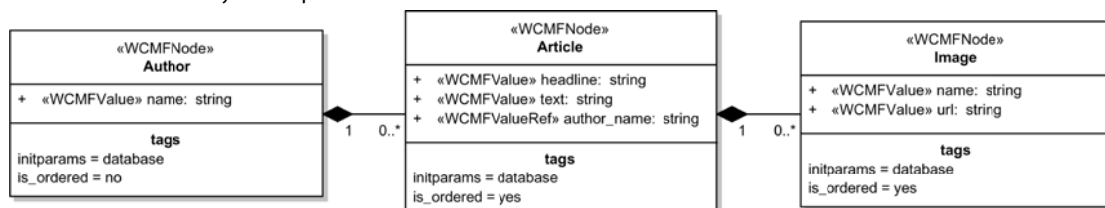


Figure 3 Example of a data model

An example of a simple data model is shown in Figure 3. Each of the entity types *Author*, *Article* and *Image* is modeled as WCMFNode stereotypes, their attributes as WCMFValue stereotypes. The only exception is the attribute *author_name*, which is a reference to the *Author's name* attribute and therefore is of the stereotype WCMFValueRef. As the relations between the classes show, one *Author* can own several *Articles* and one *Article* in turn can contain several *Images*.

INHERITANCE

If different entity types should have the same attributes, one may define a common base class that makes these available to inheriting classes. An example may be a `EntityBase` type, which defines meta information like creation date and last modification date for all entity types (see Figure 4).

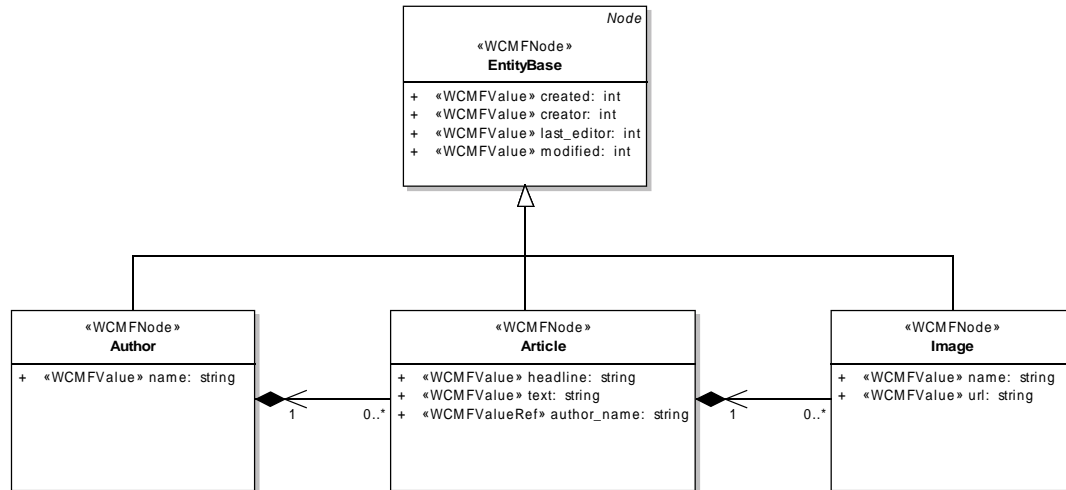


Figure 4 Example of a data model using inheritance

The generator generates table and class/mapper definitions for each class in the model. In case of inheritance the subclasses will contain the attributes of the parent classes too. The pattern used for mapping the inheritance hierarchy to the database is *concrete table inheritance*¹¹.

ASSOCIATIONS

Associations are an important concept in wCMF, since they describe the relations between entities, namely the parent-child relation. There are three different types of associations supported: Composite, shared and one-directional normal associations (see Figure 5).

It is important to understand how wCMF recognized **parents** and **children** in an association. In the case of composite and shared associations the parent is always the class, that is connected with the diamond end of the association. Normal associations are treated as parent-child relation, if they are only navigable in one direction, which is then defined as the child to parent direction.

Associations differ in the treatment of children in case of deletion of the parent. In shared and normal associations children are not deleted, while composite children are deleted on parent deletion. On the other hand the default wCMF application allows creation of child nodes in an composite association, while the other two only allow association of existing nodes. Note, that the multiplicity of children is taken into account when adding children. For example a 2 at the childs end means that the maximum number of children to add is two.

Another aspect of associations is the use of arrow ends that describe the **navigability**. If an arrow end is omitted on one end, that means that the appropriate entity type is not seen from the other end.

Regarding the database model all three association types will result in creation of a foreign key column in the child table that points to the parent table.

This association information will be written into the generated mapper classes (see 4.4).

¹¹ In contrast to the original pattern, the generator also generates tables for abstract classes. See <http://www.martinfowler.com/eaCatalog/concreteTableInheritance.html>

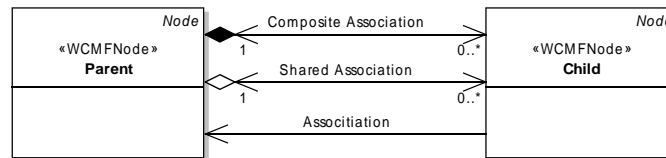


Figure 5 Supported association types

Association types

In an wCMF application the different association types shown in Figure 5 result in different behaviours regarding the operations allowed on child entities.

Association type	Delete children	Create children	Associate children
Composite association	Yes	Yes	No
Shared association	No	Yes	Yes
Association	No	No	Yes

Multiple associations

Another question that may arise is how to model **multiple associations** between two types. Given that you have *project* and *developer* entity types and you want to associate developers with projects. To achieve this, you'll draw a shared association between them and place the diamond end at the project class. But what if you want to assign one developer as project leader. In this case you'll have to draw a second association (with multiplicity 1 at both ends). The question is how can wCMF distinguish between a developer being a normal member and the project leader. This is, where the concept of roles comes into play. To resolve this conflict, you'll have to assign role names to the developer association ends. The generator will generate different entity and mapper classes for each role, which allows wCMF to know which role a developer has in relation to a given project.

DEFINITION OF THE USER INTERACTION

Here we also want to give a short overview on how to model the presentational layer.

The user interaction of a wCMF application follows the model-view-controller pattern. This means that the user is provided with HTML pages (*views*), which is in turn used to change the application data (*model*).

The Smarty Template Engine (<http://smarty.php.net>) is used to display the views.

PHP classes (*controller*) perform tasks like storing changed data and displaying views.

A controller can have none, one or more views assigned.

The control of the application, i.e. the selection of controllers and views, takes place in the class `ActionMapper`. Simplified this proceeds along the following scheme:

1. A starting point view *ViewA* assumed, which is displayed by a controller *ControllerA*. The view is a HTML page with an input form, which contains application data and buttons for different actions.
2. The user changes some data and selects action *ActionA* by pressing the appropriate button. By this all data contained in the form is sent to the `ActionMapper`.
3. With the performed action and the last executed controller the `ActionMapper` chooses the next controller *ControllerB* from the configuration file and executes it after passing the data to it.

4. The controller processes the form data. If the controller for instance has no view assigned, it automatically returns upon termination to the `ActionMapper` with a predefined action.
5. The `ActionMapper` determines the next controller (*target controller*) and executes it.

Therefore a wCMF application consists of a sequence of **actions**, which call different **controllers** and their assigned **views**. With each action the user chooses a branch, which the application flow follows. Since it is possible, that a controller should be used in different parallel application branches, the context as a third parameter - along with the last controller and action - is provided for determining the target controller. Via the context one of the application's branches can be defined. This is illustrated in the following example:

Assumed there is a `SaveController` (without a view) for storing data, which returns with the action `ok` upon finishing. The application should show the view for editing the author's data (the corresponding controller might be the `AuthorController`). To store the changed author data the user chooses the action `save`, which causes the `ActionMapper` to execute the `SaveController`. Furthermore in the configuration file could be defined, that after the action `ok` of the `SaveController`s the `AuthorController` should follow again. So far, so good.

But what would happen, if we also want to use the `SaveController` to store the article data? Then the `ActionMapper` should execute the `SaveController` in order to store them. Again the `SaveController` would return the action `ok`, which in turn causes the `ActionMapper` to execute the `AuthorController`. That certainly would be confusing.

Therefore in such a case a context has to be defined. The context is passed from controller to controller and is additionally used to determine the target controller. When an author is edited for instance the context would be `author`, whereas it would be `article` when an article is edited. By that the `ActionMapper` can always find the appropriate target controller for the `SaveController`.

To summarize the following elements contribute to the definition of the user action:

1. *Views* display the application data as HTML (input form). The user interacts with the views while he changes data and calls actions.
2. *Controllers* execute the actions while loading data, storing data, exporting data etc. In addition they handle the views.
3. *Actions* establish the connection between the controllers.
4. *Contexts* define different application branches.

Controllers are defined in PHP classes, views are Smarty templates, the definition of actions and contexts finally is made in the configuration file (`config.ini`) of the application¹². For modeling these elements the wCMF UML profile provides various stereotypes and tagged values.

STEREOTYPES FOR MODELING THE USER INTERACTION

The following stereotypes for modeling the user interaction are known to the generator:

Name	UML meta class	Meaning	Example
WCMFController	Class	Controller. WCMFController must inherit from the framework class <code>Controller</code> .	<code>LoginController</code>
WCMFView	Class	View, which is assigned to a controller	<code>login</code>
WCMFActionKey	Association	An action key used in WCMF. An action key associates two controllers (to define a control flow) or a view with a controller (to define a view attachment). Controller, Views and Associations define the application flow.	

¹² In order to create a working application - views and controllers obviously must take the defined actions and contexts into account.

For each stereotype tagged values are defined, which are shown in the following table:

Stereotype	Tagged value	Meaning	Values	Default value
WCMFController				
WCMFView				
WCMFActionKey	action	The action, which is triggered by this association		
	context	The context, in which this association is valid		
	config	The configuration file, in which this association will be placed		config.ini

The declaration of the tagged values *action* and *context* is optional. When they're not declared, it basically means "after any action do ..." and "in any context do ...".

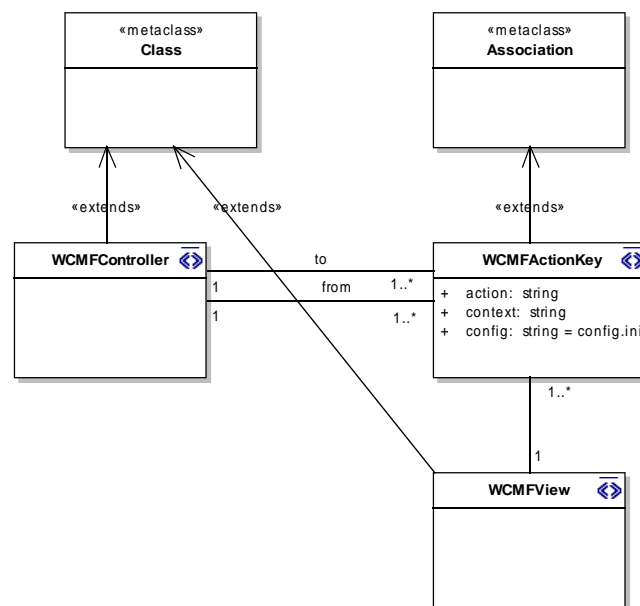


Figure 6 Part of the wCMF UML Profile (modeling the user interaction)

Figure 6 shows the part of the wCMF UML profile, which is relevant for modeling the user interaction. The stereotypes WCMFController and WCMFView are derived from the UML meta class *Class*, WCMFActionKey from *Association*. WCMFActionKeys are directional and connect two WCMFControllers or one WCMFView and one WCMFController, respectively. After one WCMFController several WCMFControllers can follow (with varying actions and/or contexts), as well as one WCMFController can have several WCMFViews assigned (with varying actions and/or contexts). Furthermore one WCMFView can be assigned to several WCMFControllers.

An example for a simple interaction model is given in Figure 7. The controllers are of the stereotype WCMFController, the relations of the stereotype WCMFActionKey. Each association's tagged values is described in a note. Since the association between *AuthorController* and the view *author* doesn't have the tagged values *action* and *context*, *AuthorController* displays this view for every action and context. The same applies to the *ArticleController* and the view *article*. The *SaveController* doesn't display a view; it only stores the changed data passed to it.

For associations, which link controllers, at least the tagged value *action* is set. Since obviously all *save* actions result in the execution of the *SaveControllers*, no context is necessary¹³. The

¹³ To simplify matters, it would also have been possible to define an association with the action *save* between the base class *Controller* and the *SaveController*. In our convention this means, that every *save* action from any controller would have the *SaveController* as target.

context is helpful, when we want to solve the ambiguousness of the action *ok* upon termination of the *SaveControllers*. We define, that the context is *author* when editing the author and *article* when editing articles. By this the *ActionMapper* always knows which controller it should return to after executing the *SaveControllers*.

For the simplifying modelling the following default rule regarding actions applies. If a controller has no view attached (meaning that execution of the following controller needs no user decision) and no *WCMFActionKey* association starts from that controller, the *ActionMapper* automatically

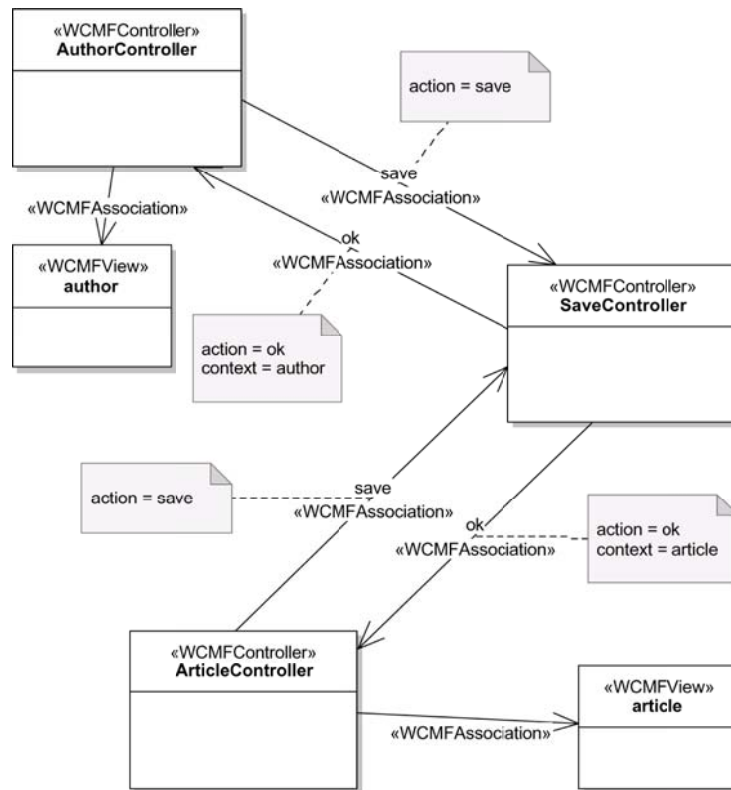


Figure 7 Example for an interaction model

chooses the preceding controller to execute next. For that reason the *ok* actions may be omitted in the diagram.

STEREOTYPES FOR MODELING THE APPLICATION CONFIGURATION

The following stereotypes for modeling the application configuration are known to the generator:

Name	UML meta class	Meaning	Example
WCMFSystem	Class	A system configuration section	Database

For each stereotype tagged values are defined, which are shown in the following table:

Stereotype	Tagged value	Meaning	Values	Default value
WCMFSystem	platform	The platform to which the configuration settings apply	e.g. wcmf	
	config	The configuration file where the settings will be placed		config.ini

The *WCMFSystem* stereotype allows to model every aspect of the configuration files. You should

one class for each section. The attributes of the class make the configuration keys while their initial values are the configuration values. The model in Figure 8 shows the complete configuration of the wCMF default application.

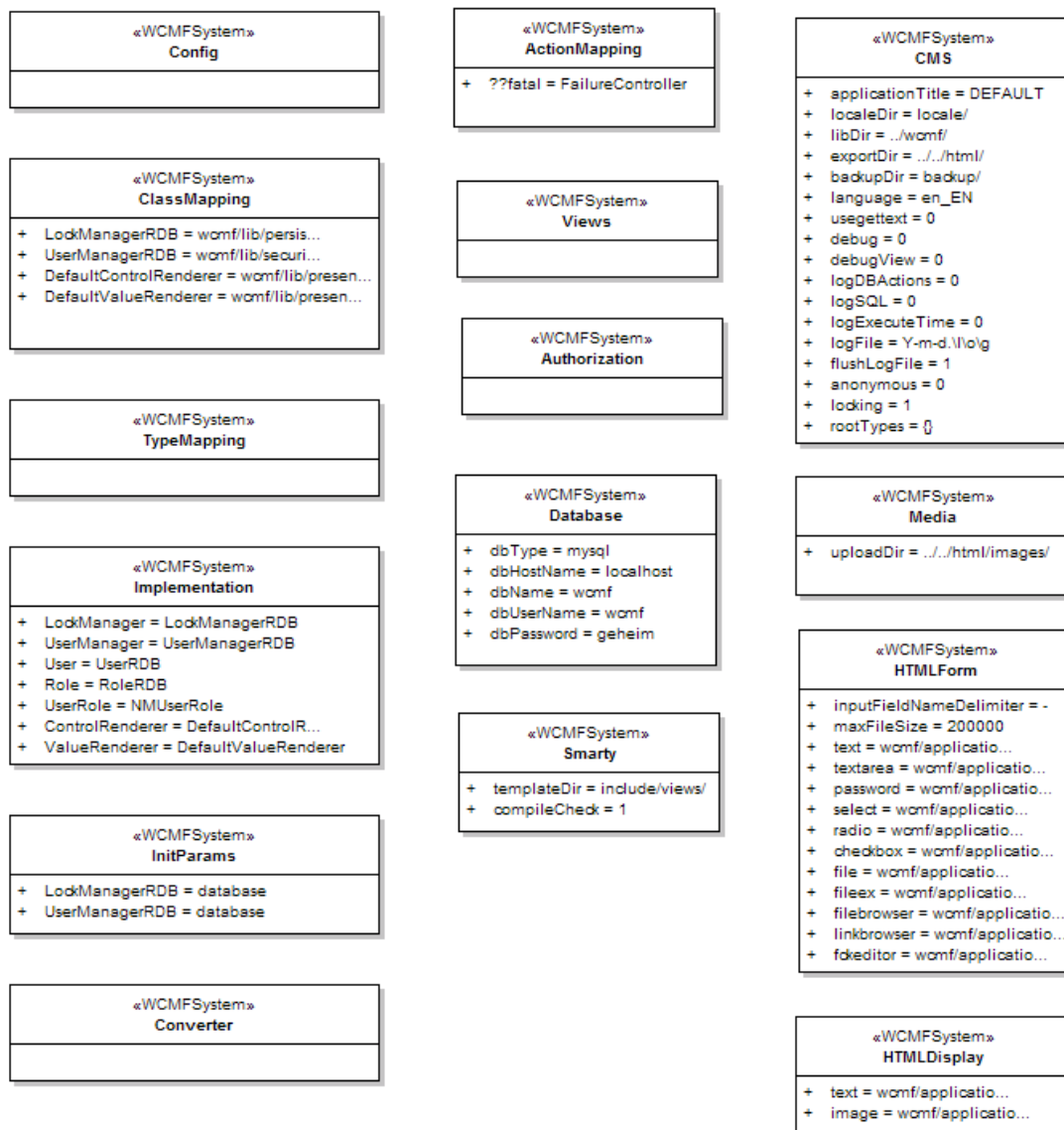


Figure 8 Example for an configuration model

4 USING THE GENERATOR

4.1 MODELING THE APPLICATION

The modeling of the wCMF application is done in an UML modeling tool. This must support the export of the model to XMI files.

The wCMF UML profile is an XMI file (wcmf_uml_profile.xml) in the directory `metamodel` of the generator installation. If Enterprise Architect is used as modeling tool, the file can directly be imported. Otherwise the profile should be created in an appropriate tool. If necessary the stereotypes and tagged values can also be assigned manually to the elements.

The model of the wCMF default application in the Enterprise Architect (wcmf_default_ea.eap) and

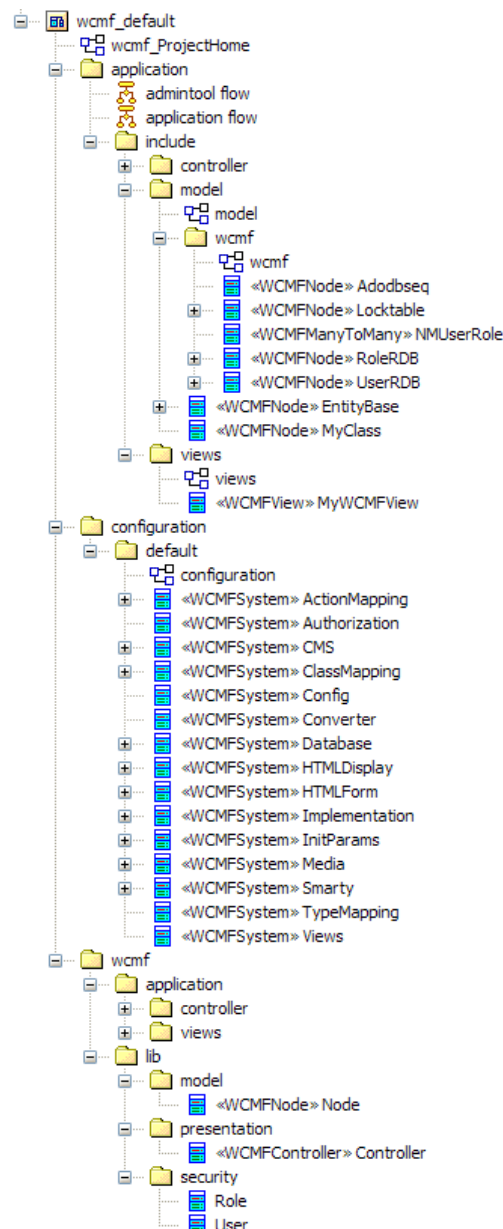


Figure 9 Package structure of the wCMF default application

the XMI format (wcmf_default_ea.xmi) can be found in the directory `model`. It can be used as basis for custom applications. Figure 9 shows the package structure of the wCMF default application¹⁴.

The package `wcmf` contains the framework classes, which are used for modeling¹⁵:

- It contains the framework's controller (in the package `wcmf/application/controller`) and views (in the package `wcmf/application/views`), which can be reused in the customized application,
- as well as the classes `Node` and `Controller` (in the package `wcmf/lib/model`, `wcmf/lib/presentation`, respectively), which are used as base classes for custom WCMFNodes and WCMFController.

¹⁴ If the model can't be imported by the tool, it can be rebuild.

¹⁵ The package structure corresponds to the directory structure of the wCMF framework.

- The package `wcmf/lib/security` contains the classes `User` and `Role`, from which user and role implementation classes must inherit. For convenience there are already implementation classes added to the model (`UserRDB` and `RoleRDB` in the package `application/model/wcmf`) from which the generator will create appropriate classes¹⁶.

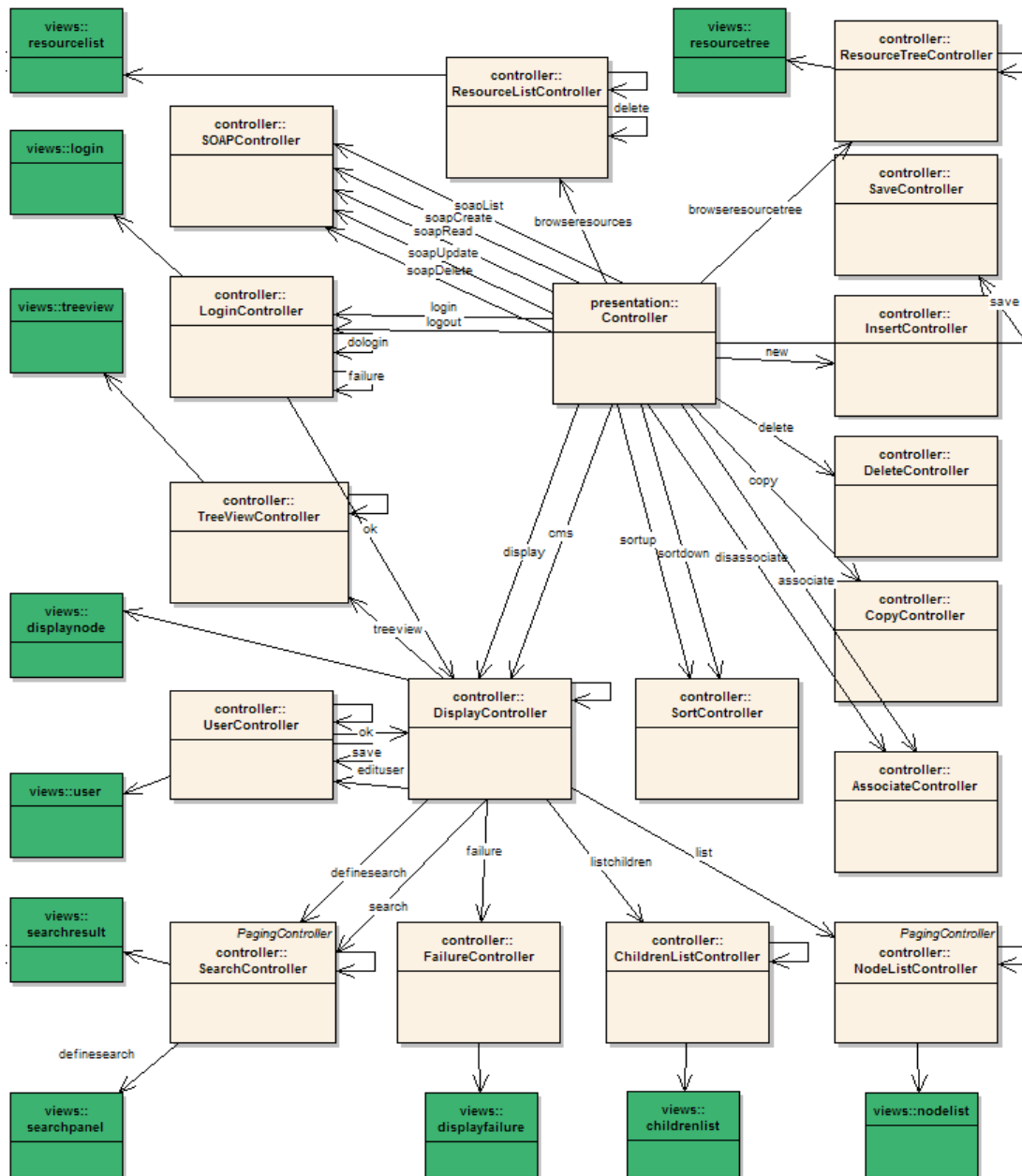


Figure 10 Model of the user interaction of the default application

The custom application is to be modeled in the package `application`. Although it has proved as suitable in many projects, it's not necessary to use the package's structure. On the top level there are two diagrams, which show the user interaction of the default application (Figure 10) and the admintool (Figure 11). These two applications are completely based on the classes contained in the framework. Although they are both completely modeled, no content can be managed, as long as no data model is given. The following steps can be taken to create a custom application

¹⁶ The package `application/model/wcmf` contains WCMFNode representations for all database tables wCMF uses by default.

based on the wCMF default application.

- *Definition of the data model:* This step is necessary. For each entity type a WCMFNode in the package `application/include/model` must be created. Figure 12 shows the default content of the diagram. There is already a class `EntityBase` that contains attributes useful for all entity types, which may therefore be used as common base class. Another class (`MyWCMFClass`) may be used as entry point for modeling custom entity types.

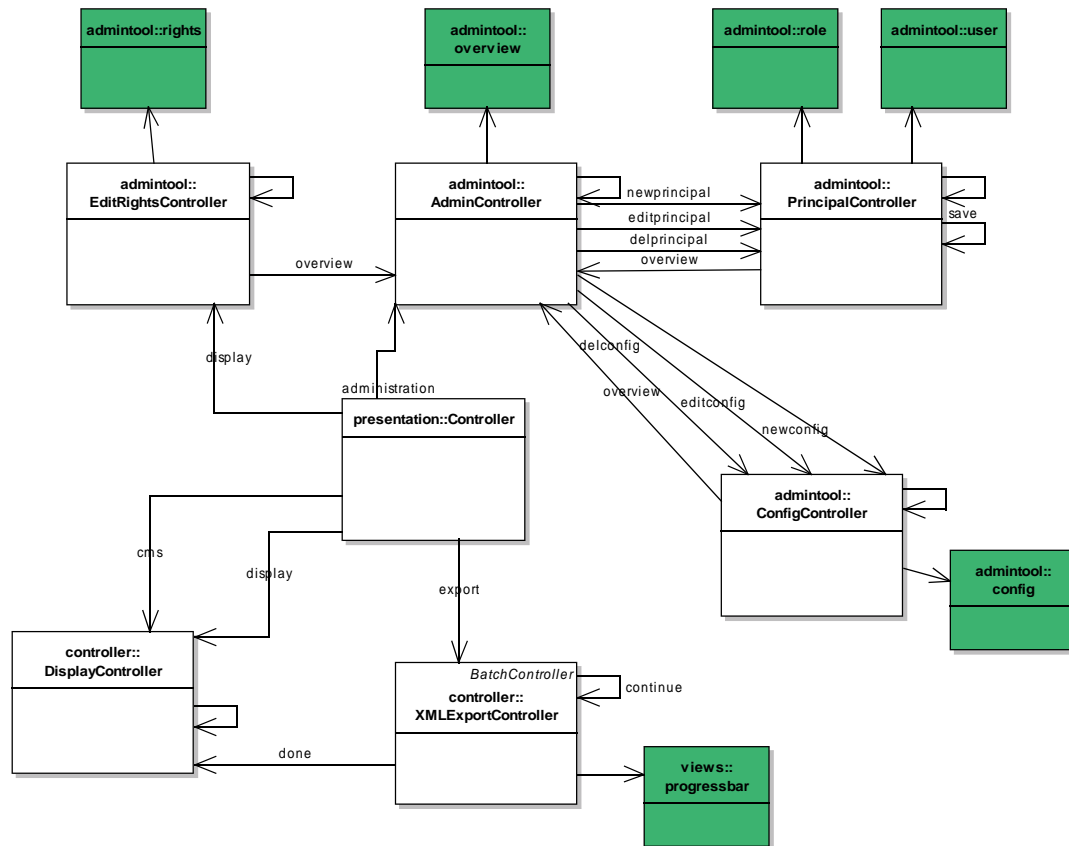


Figure 11 Model of the admintool's user interaction

To define hierarchical content structures the classes must be provided with the appropriate associations. Please note, that N to M relations between nodes are modeled by introducing an 1 to N relation on both sides¹⁷. After this step the default application is able to manage custom content.

- *Extension of the user interaction:* To adapt the interaction capabilities to particular requirements, controllers (in the package `application/include/controller`) and views (in the package `application/include/views`) can be added. Framework controllers partially define points for modification, so that their behavior can be adapted to special requirements via inheritance. Customized controllers and views can replace those in the diagrams. Furthermore new associations (for new actions) and diagrams can be created.

¹⁷ If an Article instance should contain N Images and one Image at the same time should be assigned to M Articles, the result would be a N to M relation. Since this can't be implemented using foreign keys in a relational database, a relation table must be created. A corresponding relation type (e.g. `NMArticleImage`) must be added to the wCMF data model, which has an 1 to N relation to Article and Image, respectively. It is then possible, that one Article instance contains any number of `NMArticleImage` instances (which each point to one Image) and one Image instance contains any number of `NMArticleImage` instances (which each point to one Article). For N to M entities use the `WCMFManyToMany` stereotype. The wCMF will handle these relations transparently to the user.

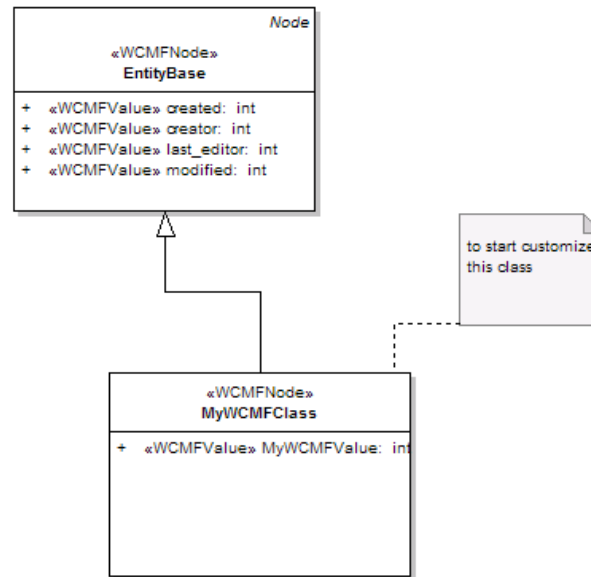


Figure 12 Start modeling

- *Customization for different users:* By using the tagged value *config* in the WCMFActionKeys it's possible to assign user interactions to different configuration files and with this to different users. An example for this is the admintool, because the configuration file admin.ini is only assigned to the administrator.

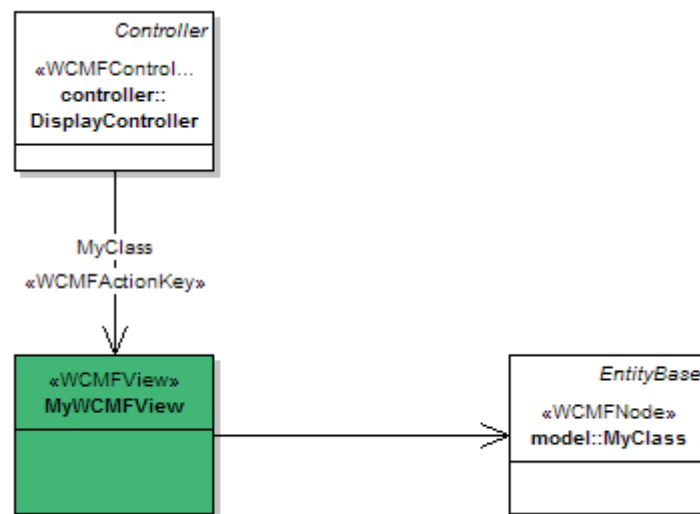


Figure 13 View customization

- *Adapting the views:* There are different ways to adapt the application views. The most simple is to change the cascading stylesheets (*.css files). If the views should be completely reworked, it is possible to attach custom views to the controllers and implement them¹⁸.

Another interesting method for view customization is depicted in Figure 13. The wcmf default application shows the same generic editing view for every entity type. Let us assume that a view for a particular entity should be customized (e.g. one attribute should be hidden). This is easily achieved by connecting a new view to `DisplayController`. The connection is drawn

¹⁸ Smarty templates that are included other views and therefore do not appear in the model may be overridden in the application's view directory. For example if only the navigation template of the default application should be customized it suffices to copy `navigation.tpl` from `wcmf/application/views/` to `application/include/views/application/views/` and modify it. Please note that the target directory mirrors the wcmf view directory inside the application view directory. This allows the framework to automatically find the right view.

with a `WCMFActionKey`, whose tagged value `context` is set to the name of the entity to display. This makes the application select this view, when an instance of that entity type is displayed. To tell the generator, which input fields to generate into the view, the entity type has to be connected to the view. Due to this the generator creates an editing form for the entity type, that may be customized later.

- *Customization of the principal classes:* By inheriting from `UserRDB` and `RoleRDB` it is possible to add custom attributes to the principal classes. The custom classes must be referenced in the configuration file in the `implementation` section.

After the application is modeled, the generator can be started. Of course it's possible to change the model afterwards and start the generator again. Source code, which was changed by hand will be preserved and not overwritten (see 4.4).

4.2 CONFIGURATION OF THE GENERATOR

The configuration of the generator is stored in a project file. The structure of the file follows the scheme of Java property files, i.e. the configuration parameters are stored as key-value pairs. An example configuration can be found in the main directory of the generator installation. According to the convention it has the file extension *properties*.

The following table shows the meaning of the individual configuration parameters¹⁹.

Parameter name	Meaning	Default value
<code>de.bmiag.genfw.logger.level</code>	Level of detail of log messages	3
<code>de.bmiag.genfw.ui.class</code>	The used generator interface. There is a text and an interactive mode.	<code>de.bmiag.genfw.ui.swing.SwingUI</code>
<code>de.bmiag.genfw.textui.select</code>	Name of the meta class, for which the generator expands the root template	Model
<code>de.bmiag.genfw.textui.check</code>	Declares, whether the check template should be expanded before generation	true
<code>de.bmiag.genfw.baseuml.identifier.class</code>	Implementation of the default identifier	<code>com.wemove.wcmf.meta.Identifier</code>
<code>de.bmiag.genfw.instantiator.class</code>	Implementation of the model instantiator	<code>com.wemove.wcmf.meta.Instantiator</code>
<code>de.bmiag.genfw.instantiator.xmlmap</code>	The XMI specification used for the model import	<code>mappings/rose_unisys132_xmi10_all_wcmf.xml</code> ²⁰
<code>de.bmiag.genfw.instantiator.tooladapter.class</code>	The tool adapter used for the model import	<code>com.wemove.wcmf.toolsupport.WCMFRoseAdapter</code> ²¹
<code>de.bmiag.genfw.instantiator.design</code>	Model of the wCMF application	<code>model/wcmf_default_ea.xmi</code>
<code>de.bmiag.genfw.instantiator.metamap</code>	Mapping of stereotypes to meta model classes	<code>mappings/metamappings.xml</code>
<code>configfile.default</code>	Name of the standard configuration file. General properties like the type mapping are declared here.	<code>config.ini</code>
<code>library.package</code>	Model package, in which the wCMF framework is defined. From this package no files will be generated.	wcmf
<code>application.package</code>	Model package, in which the application is modeled	application

¹⁹ Further explanations can be found in *openArchitectureWare's* documentation, since most of the parameters refer to properties of the generator framework.

²⁰ When using Poseidon (3.x) set this to `de/bmiag/genfw/instantiator/xml/toolsupport/baseuml/poseidon/poseidon30_xmi12_all.xml`

²¹ When using Poseidon (3.x) set this to `de.bmiag.genfw.instantiator.xml.toolsupport.baseuml.poseidon.Poseidon3Adapter`

Parameter name	Meaning	Default value
root.package	The package that contains the library and application package (if any). It's name will be stripped from paths to classes ²²	wcmf_default
projectname	Name of the application	wcmf_default
de.bmiag.genfw.xpand.path	Path to the generator templates	templates/with_domain_classes
de.bmiag.genfw.presolver.path	Directory, from which the protected regions are read	preresolve
backup.dir	Directory, in which the backup of the existing application is stored before generation. If the value is empty, no backup will be made ²³ .	backup
de.bmiag.genfw.filewriter.path	Directory, in which the application is created ²⁴ .	target_ea
framework.file	The zip file, from which the framework should be extracted (no action if the value is empty)	wcmf/wCMF.zip
xslt.stylesheet	Xslt documentation generation from xmi file (if xslt.output is empty no documentation will be created) multiple files maybe given in a list separated by ';' (ms windows) or ':' (unix)	xsl/wCMFDoc.xslt
xslt.output	The documentation target file including the path	target_ea/application/documentation.html

The given values for the import 's configuration are those working for Enterprise Architect. If another modeling tool is used the values might have to be changed. The generator framework provides some import definitions and tool adapters.

The generator provides a default set of templates in the `templates/with_domain_classes` directory. When these are used, the generator will create one class with the corresponding name for each entity type of the data model. The class inherits from `Node` and has specialized getter- and setter-methods for the content attributes and associated entity types. The `Persistent-Mapper` classes, which are generated as well, inherit from `NodeUnifiedRDBMapper` and create instances of the entity classes.

4.3 RUNNING THE GENERATOR

The generator is started with the command

```
java -jar wCMFGenerator.jar configfile
```

The parameter *configfile* is to be replaced with the name of the configuration file to be used. Subsequently the following steps are carried out:

1. If the application directory already exists and an backup directory is set, an archive with the application directory's content will be stored in the backup directory. The name of the archive is composed of the application name and the date and time. If no application directory exists, it will be created.
2. In the second step the generator creates the `preresolve` directory (if not existing) and copies the files of the application directory to it. The files in this directory are later used as reference for the generator to create the content of the `PROTECTED REGIONS`, which are contained in the generator templates. If code was manually added in those sections of the application files, it will be preserved after generation. Each region has an ID which must be unique.
3. After that the generator interface is started in interactive mode (see Figure 14). It displays a tree structure of the model. In this view the user can check if all elements were recognized correctly.

²² Enterprise Architect only allows to put packages into a view (which is interpreted as a parent package on generation). To get rid of this, set the `root.package` property to the name of the view.

²³ The content of the directory `de.bmiag.genfw.filewriter.path/application.package` will be backed up.

²⁴ This path expects a normal slash "c:/temp". If you use a backslash like "c:\temp" the generator will not work.

Selecting *Project/Check Selected* from the menu checks if the selected part of the model is modeled correctly (e.g. WCMFControlller do inherit from the framework class Controller)²⁵.

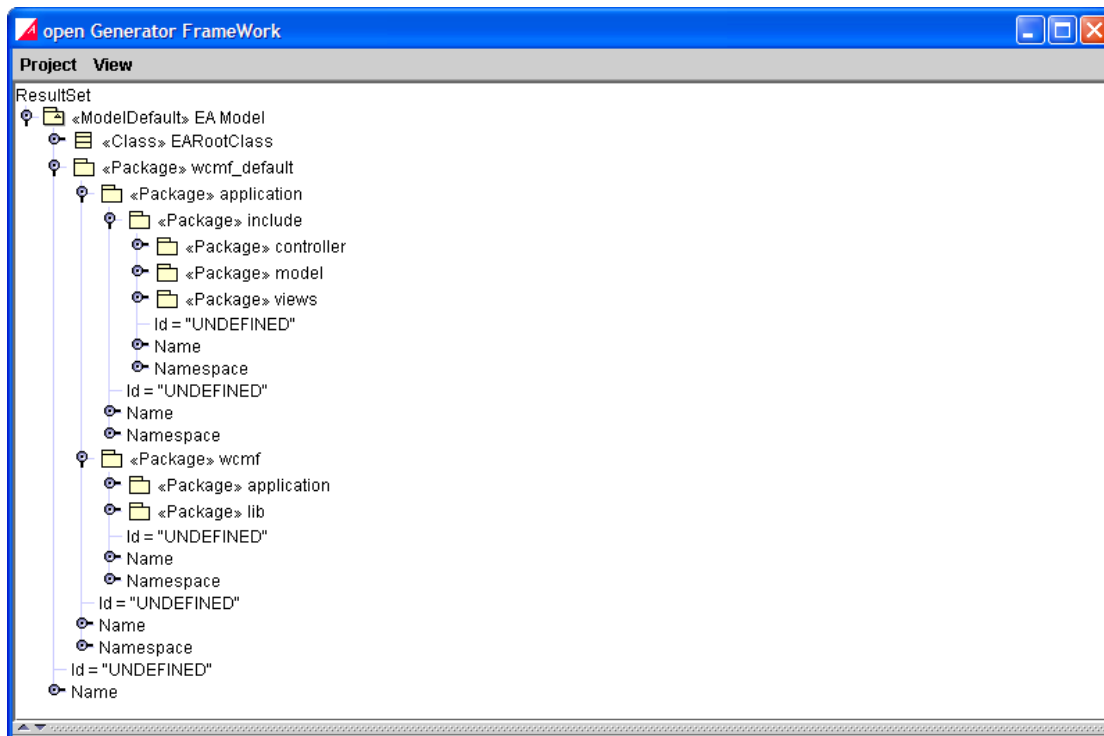


Figure 14 WCMFGenerator in interactive mode

- Now the actual generation is done. The generation is started by selecting *Project/Generate Selected* from the menu. The part of the tree, which should be generated, has to be selected (it's recommended to select the node *Model* itself). The generator will now create the application files from the templates and the model. Manually added code in the PROTECTED REGIONS is preserved.

When the text interface is used, the individual steps are processed automatically. This makes an easy integration into other tools possible.

Since the generator only saves the PROTECTED REGIONS upon start – and not before every generation – changes, which were made while the generator is open, are lost.

4.4 GENERATION RESULT

The generator creates the following files²⁶:

- SQL Scripts:** In the directory `application/install` the generator creates a file called `tables.sql`. This file contains the MySQL DDL scripts for creating the database tables for the different entity types (WCMFNodes). Each entity type has a table assigned, which makes the integration of the tables into other applications easier. Each table contains a primary key named `id`²⁷. To realize the relations between entity types a foreign key with the name `fk_parentType_id`²⁸ is put into each dependent table. Entity types, which can be sorted, get the additional attribute `sortkey`.

For the type *Article* from the example above the generator creates the following code:

²⁵ Please note, that during this step the generator also checks if the IDs of the PROTECTED REGIONS are unique and stops if they are not.

²⁶ The examples apply to the default settings (e.g. the provided generator templates are used, application can be found in the package `application`). Besides all results are based on the provided generator templates.

²⁷ Use tagged value `pk_name` of WCMFNode stereotype to define a different name or use compound primary keys.

²⁸ Use tagged value `fk_name` of WCMFRelationEnd to define a different name.

```

#
# Structure of Table `Article`
#
# version 1.0
#
DROP TABLE IF EXISTS `Article`;
CREATE TABLE `Article` (
  `id` INT(11) NOT NULL,
  `fk_author_id` INT(11), # references Author
  `headline` VARCHAR(255),
  `text` VARCHAR(255),
  `created` VARCHAR(255),
  `creator` VARCHAR(255),
  `last_editor` VARCHAR(255),
  `modified` VARCHAR(255),
  `sortkey` INT(11),
  PRIMARY KEY (`id`)
) TYPE=MyISAM;

```

Inheritance is resolved by adding the base class attributes to every table of the inheriting types (*Concrete Table Inheritance*).

- *PersistentMapper*: In the directory `application/include/model` the generator creates the mapper files – one for each entity type. The mappers inherit from the class `NodeUnifiedRDBMapper` and implement the demanded methods.

The following code section shows the class `ArticleRDBMapper` (defined in the file `class.ArticleRDBMapper.php`):

```

class ArticleRDBMapper extends NodeUnifiedRDBMapper
{
  /**
   * @see RDBMapper::getType()
   */
  function getType()
  {
    return 'Article';
  }
  /**
   * @see NodeRDBMapper::createObject()
   */
  function &createObject($oid=null)
  {
    return new Article($oid);
  }
  /**
   * @see NodeUnifiedRDBMapper::getTableName()
   */
  function getTableName()
  {
    return 'Article';
  }
  /**
   * @see PersistenceMapper::getPkNames()
   */
  function getPkNames()
  {
    return array('id' => DATATYPE_IGNORE);
  }
  /**
   * @see NodeUnifiedRDBMapper::getMyFKColumnNameImpl()
   */
  function getMyFKColumnNameImpl($parentType)
  {
    if ($parentType == 'Author') return 'fk_author_id';
    if ($this->getType() == 'Article' && $parentType == 'Author') return 'fk_author_id';
    return '';
  }
  /**
   * @see NodeUnifiedRDBMapper::getOrderBy()
   */
  function getOrderBy()
  {
    return array();
  }
  /**
   * @see NodeUnifiedRDBMapper::getObjectDefinitionImpl()
   */
  function getObjectDefinitionImpl()
  {
    $nodeDef = array();
    $nodeDef['_properties'] = array
    (
      array('name' => 'is_searchable', 'value' => true),
    );
  }
}
// PROTECTED REGION ID(application/include/model/class.ArticleRDBMapper.php/Properties) START
// PROTECTED REGION END
);

```

```

$nodeDef['_datadef'] = array
(
    // Value description:
    array('name' => 'id', 'app_data_type' => DATATYPE_IGNORE, 'column_name' => 'id',
        'db_data_type' => 'INT(11) NOT NULL', 'default' => '', 'restrictions_match' =>
        '', 'restrictions_not_match' => '', 'restrictions_description' => '',
        'is_editable' => false, 'input_type' => 'text', 'display_type' => 'text'),
    // Value description:
    array('name' => 'fk_author_id', 'app_data_type' => DATATYPE_IGNORE, 'column_name' =>
        'fk_author_id', 'db_data_type' => 'INT(11)', 'default' => '',
        'restrictions_match' => '', 'restrictions_not_match' => '',
        'restrictions_description' => '', 'is_editable' => false, 'input_type' =>
        'text', 'display_type' => 'text'),
    // Value description:
    array('name' => 'headline', 'app_data_type' => DATATYPE_ATTRIBUTE,
        'column_name' => 'headline', 'db_data_type' => 'VARCHAR(255)',
        'default' => '', 'restrictions_match' => '',
        'restrictions_not_match' => '',
        'restrictions_description' => '', 'is_editable' => true,
        'input_type' => 'text', 'display_type' => 'text'),
    ...

    // Value description:
    array('name' => 'modified', 'app_data_type' => DATATYPE_ATTRIBUTE, 'column_name' =>
        'modified', 'db_data_type' => 'VARCHAR(255)', 'default' => '',
        'restrictions_match' => '', 'restrictions_not_match' => '',
        'restrictions_description' => '', 'is_editable' => true, 'input_type' =>
        'text', 'display_type' => 'text'),
    // Value description: Sort key for ordering
    array('name' => 'sortkey', 'app_data_type' => DATATYPE_IGNORE, 'column_name' => 'sortkey',
        'db_data_type' => 'INT(3)', 'default' => '', 'restrictions_match' => '[0-9]*',
        'is_editable' => true, 'input_type' => 'text[class="tiny"]');
);
$nodeDef['_ref'] = array
(
    // Value description:
    array('name' => 'author_name', 'ref_type' => 'Author', 'ref_value' => 'name', 'ref_table'
        => 'Author', 'id_column' => 'id', 'fk_columns' => 'fk_article_id', 'ref_column'
        => 'name');
);
$nodeDef['_parents'] = array
(
    array('type' => 'Author', 'is_navigable' => true, 'table_name' => 'Author',
        'pk_columns' => array('id'), 'fk_columns' => 'fk_author_id');
);
$nodeDef['_children'] = array
(
    array('type' => 'Image', 'minOccurs' => 0, 'maxOccurs' => 'unbounded', 'aggregation' =>
        true, 'composition' => true, 'is_navigable' => false, 'table_name' => 'Image',
        'pk_columns' => array('id'), 'fk_columns' => 'fk_article_id', 'order_by' =>
        array('sortkey'));
);
return $nodeDef;
}
}

```

Note, that the mappers node definition also contains the attributes from the base class *EntityBase*.

- *Domain classes*: If the generator templates in the directory `templates/with_domain_classes` are used, the generator creates one domain class for each *WCMFNode* in the directory `application/include/model`. The following example shows the domain class of the type *Article* (defined in the file `class.Article.php`)²⁹:

```

class Article extends EntityBase
{
    // PROTECTED REGION ID(application/include/model/class.Article.php/Member) START
    // PROTECTED REGION END
    function Article($oid=null, $type=null)
    {
        if ($type == null) parent::EntityBase($oid, 'Article');
        else parent::EntityBase($oid, $type);
    }
    // PROTECTED REGION ID(application/include/model/class.Article.php/Construct) START
    // PROTECTED REGION END
    /**
     * @see PersistentObject::validateValue()
     */
    function validateValue($name, $value, $type=null)

```

²⁹ The generator names the getter- und setter methods according to the scheme *getAttributeName* and *setAttributeName*. To avoid an unintentional overwriting of methods of the base class *Node* and *PersistentObject*, the attribute names should be chosen with care or the generator templates should be changed accordingly. This means for instance that the respective methods of the base classes would be overwritten, if attribute names like *type* or *state* are used.

```

    {
        $result = true;
// PROTECTED REGION ID(application/include/model/class.Article.php/validateValue) START
// PROTECTED REGION END
        return $result;
    }
/**
 * @see PersistentObject::getObjectDisplayName()
 */
function getObjectDisplayName()
{
    return Message::get("Article");
}
/**
 * @see PersistentObject::getObjectDescription()
 */
function getObjectDescription()
{
    return Message::get("Article represents an article written by an author.");
}
/**
 * @see PersistentObject::getValueDisplayName()
 */
function getValueDisplayName($name, $type=null)
{
    $displayName = $name;
    if ($name == 'id') $displayName = Message::get("id");
    if ($name == 'fk_author_id') $displayName = Message::get("fk_author_id");
    if ($name == 'headline') $displayName = Message::get("headline");
    if ($name == 'text') $displayName = Message::get("text");
    return Message::get($displayName);
}
/**
 * @see PersistentObject::getValueDescription()
 */
function getValueDescription($name, $type=null)
{
    $description = $name;
    if ($name == 'id') $description = Message::get("");
    if ($name == 'fk_author_id') $description = Message::get("");
    if ($name == 'headline') $description = Message::get("The articles headline");
    if ($name == 'text') $description = Message::get("The article text");
    return Message::get($description);
}
/**
 * See if the node is an association object, that implements a many to many relation
 */
function isManyToManyObject()
{
    return false;
}
/**
 * Getter/Setter
 */
function getId($unconverted=false)
{
    if ($unconverted) return $this->getUnconvertedValue('id', DATATYPE_IGNORE);
    else return $this->getValue('id', DATATYPE_IGNORE);
}
function setId($id)
{
    return $this->setValue('id', $id, DATATYPE_IGNORE);
}
function getFkAuthorId($unconverted=false)
{
    if ($unconverted) return $this->getUnconvertedValue('fk_author_id',
        DATATYPE_IGNORE);
    else return $this->getValue('fk_author_id', DATATYPE_IGNORE);
}
function setFkAuthorId($fk_author_id)
{
    return $this->setValue('fk_author_id', $fk_author_id, DATATYPE_IGNORE);
}
function getHeadline($unconverted=false)
{
    if ($unconverted) return $this->getUnconvertedValue('headline',
        DATATYPE_ATTRIBUTE);
    else return $this->getValue('headline', DATATYPE_ATTRIBUTE);
}
function setHeadline($headline)
{
    return $this->setValue('headline', $headline, DATATYPE_ATTRIBUTE);
}
function getText($unconverted=false)
{
    if ($unconverted) return $this->getUnconvertedValue('text', DATATYPE_ATTRIBUTE);
    else return $this->getValue('text', DATATYPE_ATTRIBUTE);
}
function setText($text)
{
    return $this->setValue('text', $text, DATATYPE_ATTRIBUTE);
}
}

```

```

function getSortkey()
{
    return $this->getValue('sortkey', DATATYPE_IGNORE);
}
function setSortkey($sortkey)
{
    return $this->setValue('sortkey', $sortkey, DATATYPE_IGNORE);
}
function getAuthorName()
{
    return $this->getValue('author_name', DATATYPE_ATTRIBUTE);
}
function getAuthorOID()
{
    $fkValue = $this->getValue('fk_author_id', DATATYPE_IGNORE);
    if ($fkValue != null) return PersistenceFacade::composeOID(array('type' =>
        'Author', 'id' => array($fkValue)));
    else return null;
}
function setAuthor(&$node)
{
    if ($node != null) $node->addChild($this);
}
function getImageList()
{
    return $this->getChildrenEx(null, 'Image', array('fk_article_id' =>
        $this->getDBID()), null);
}

...

// PROTECTED REGION ID(application/include/model/class.Article.php/Body) START
// PROTECTED REGION END
}

```

There are several PROTECTED REGION tags, into which code can be manually inserted. Future generator runs preserve this code. Please note, that in case of the class definition, the base class attributes are not taken into account. They are defined in the base class definition, which will also be generated. Besides getter and setter methods for attributes and parent and child nodes there are also methods, which return information about the entity object, which is taken from the UML model descriptions and will be translated using the `Message::get()` method.

- *Configuration files:* The generator creates the default configuration file `config.ini` in the directory `application/include`, into which general settings like the definition of the entity types are written. In the same directory the generator creates one file for each additional configuration defined in the model (via the tagged value `config` of the stereotype `WCMFActionKey`). The following section shows the configuration of the type `Article` and the user interaction defined in the example above:

```

[classmapping]
SaveController = wcmf/application/controller/class.SaveController.php
AuthorController = application/include/controller/class.AuthorController.php
ArticleController = application/include/controller/class.ArticleController.php
ArticleRDBMapper = application/include/model/class.ArticleRDBMapper.php
...
[typemapping]
Article = ArticleRDBMapper
...
[initparams]
ArticleRDBMapper = database
...
[actionmapping]
AuthorController??save = SaveController
SaveController?author?ok = AuthorController
ArticleController??save = SaveController
SaveController?article?ok = ArticleController
...
[views]
AuthorController?? = application/include/views/author.tpl
ArticleController?? = application/include/views/article.tpl
...
[database]
dbType = mysql
dbHostName = localhost
dbName = wcmf_cookbook
dbUserName = wcmf
dbPasswd = geheim
...

```

The section `classmapping` contains all WCMFNodes and WCMFControllers of the model. The user interaction is defined in the section `actionmapping`. Each WCMFActionKey corresponds with an entry of the kind `EntryController?Context?Action = TargetController`, where *Context* and *Action* correspond with tagged values. If one or both values are missing, the according place is empty. In the section `views` the WCMFViews' definitions of the controllers can be found. Since these are dependent on context and action as well, the same form of declaration as in the section `actionmapping` is used.

- **Controller:** For each WCMFController defined in the model a class definition file named `class.controllername.php` is generated in the directory `application/include/controller`. As an example the *ArticleController* is listed below.

```

class ArticleController extends Controller
{
// PROTECTED REGION ID(application/include/controller/class.ArticleController.php/Body) START
/**
 * @see Controller::hasView()
 */
function hasView()
{
return true;
}
/**
 * @see Controller::executeKernel()
 */
function executeKernel()
{
return false;
}
// PROTECTED REGION END
}

```

Only the two obligatory methods `hasView` and `executeKernel` are implemented with default behaviour. Note, that the whole body of the class definition is included in a `PROTECTED REGION`, which allows the programmer to add custom code.

- **Views:** For each WCMFView of the model the generator creates a file named `viewname.tpl` in the directory `application/include/views`. Since the generator doesn't know, what's to be displayed in the view, the file merely contains a `PROTECTED REGION`, into which HTML and Smarty Code can be manually inserted.

5 MODIFICATION OF THE GENERATOR

There are various possibilities to customize the generator for special requirements:

- The generated files are based on the used generator templates (in the generator's `templates` directory). The generated code – for example which mapper base classes are used or the SQL DDL scripts – can be adapted in those templates. For fundamental changes it's important to have knowledge of the generator framework's template language (Xpand) and of the used meta model, which is implemented in Java.
- It is also possible to expand the wCMF UML Profile. To do this first of all new stereotypes and tagged values must be defined. To incorporate these in the generator, the meta model, which is used by the generator, must be adapted accordingly. The model is implemented in the package `com.wemove.wcmf.meta.model`. Knowledge in Java is required. More details can be found in the generator framework's documentation.
- Further modifications are of course possible since the source code of the generator is available. For example the actions, which are executed in the generator's start phase (backup etc.), are implemented in the class `com.wemove.wcmf.generator.Generator.java`. Here Java knowledge is required as well.