MooseDomainGenerator

Marc Stettler

April 27, 2005

Contents

Chapter 1

Abstract

Communication is the most important thing in the world today with the internet as its most potent medium. In software reengineering, tools have the problem of not being able to communicate with each other, because they do not communicate using a standard language. MOF is the OMG standard language to exchange reengineering information. This project focussed on the communication of the reengineering tool Moose with other reengineering tools by using MOF.

Chapter 2

Introduction

In recent years the once small software reengineering community has grown considerably. New tools for software analysis have popped up everywhere, each featuring new possibilities. But very few standard languages to let the tools communicate with each other have been worked out and almost every tool has its own format. When people started to work together the problems immediately started. One could not hand information about a software model from one tool to another. So standard languages like MOF have been introduced to be able to exchange such information and let the tools communicate with each other.
The project *MooseDomainGenerator* was started with that sort of communication in mind. Tools that could export information about a software model in a standard language (like MOF) should be able to exchange this information with Moose. With a meta-model description the *MooseDomainModelGenerator* could generate the meta-model based on this description and then in a next step load the actual model into Moose. With this done, Moose could then manipulate the loaded model and save it back to disk with the original description and by doing this, communicate with another tool.
As an interesting side effect one could also generate whole software meta-models by just passing a meta-model description to the *MooseDomainModelGenerator* and generating large pieces of code in an instant.
The basic technology used for this project was VisualWorks Smalltalk as the programming language and MOF for meta-model descriptions. For further studies on these technologies please visit the CINCOM homepage or the OMG homepage respectively.

Chapter 3

Meta-Model Generation

3.1 A Few Notes

To be able to automaticaly generate a meta-model a description of the meta-model is needed. Since programs in general consist of multiple interrelated parts, one not only needs the descriptions of the parts, but also the descriptions of the associations of the parts. MOF, or in the case of the project EMOF, is the language to describe software meta-models. Sadly, as nothing is perfect, we needed to make some adjustments to our needs. In particular, we added the following functionality:

MOFExtendedAttribute

- An attribute is sometimes of more use when it knows about the type it has
- An attribute should know on which level (instance, class) it is being used
- An attribute should know with which multiplicity it can be used
- An attribute has almost always an accessor method. So an attribute should know the name of its accessor method

MOFExtendedMultipleValueAttribute

- subclass of MOFExtendedAttribute
- A class to store multi valued attributes

MOFExtendedClass

- A class has to know, whether it describes a model class or an implementation class

MOFExtendedPackage

- MOFPackage was meant to be a static concept, but when generating meta-models what one really needs is a dynamic package
- Single classes together with their associations can be extracted in a new package

- Convenience methods have been added to iterate over the elements of a package

With all the functionality that EMOF and these extensions provide one can now build some highly complex structures of software models. Now that we know the basics of the technology used, we can have a more detailed look at the generation itself. (Methods not being displayed in the following sections are shown in *italic font face* and can be found in Appendix A on page 21)

3.2 Generation Overview

generateModel: aModelDescription

```
1:   self createPackage: aModelDescription name.
2:   modelDescription := aModelDescription.
3:   self compileRootEntity.
4:   aModelDescription allClassNamesdo: [:each |
5:       self classDefinitionAndMethodsFor: each in: aModelDescription]
```

The generation of a meta-model is made up of two parts. First we need to generate an environment for the meta-model about to be generated with the *MooseDomainModelGenerator* (line 1 and 3). Only then can we go on to the task of generating the meta-model (line 4 and 5). In the following short section we will discuss the generation of the environment for the meta-model and then we will look at the actual meta-model generation itself.

3.3 Environment Generation

createPackage: aSymbol

```
1:   | package |
2:   package := Store.Registry packageNamedOrCreate: aSymbol asString.
3:   Store.Registry currentPackage: package.
4:   ^package
```

computeClassRootEntityDefinition

```
1:   ^'SCG.Moose defineClass: #', self rootOfEntityHierachyClassName ,'
2:   superclass: #{SCG.Moose.', self rootInMoose ,'}
3:   indexedType: #none
4:   private: false
5:   instanceVariableNames: ""
6:   classInstanceVariableNames: ""
7:   imports: ""
8:   category: ', modelDescription name asString, "
```

Because in Smalltalk everything is organized in packages, we first have to ask the registry in the store in line 2 to create us a new package and in line 3 to select it as the active package for the following operations. This package will contain the classes of our meta-model and is named after the name of the meta-model.
The second issue is the superclass of the classes about to be generated. It

would only seem logical to choose a superclass that is in Moose because the *MOFDomainModelGenerator* is all about communication of tools with Moose and it would also seem logical that the selected class should represent meta-model elements. The class of our choice was *AbstractEntity* which is the root for all meta-model elements in Moose. Instead of setting *AbstractEntity* as the direct root of the meta-model we generate a new class which is a subclass of *AbstractEntity* and this class will be the root of the meta-model. By doing this we have the freedom to add common behavior to generated meta-model classes. To choose an intermediate class to do this has three reasons. First we do not pollute the meta-model classes by adding shared behavior and second we do also not pollute the *AbstractEntity* class. Third by doing it that way we would be able to generate meta-model dependent shared behavior (which we do not use at the moment).

3.4 Class Generation

classDefinitionAndMethodsFor: aSymbol in: aModelDesc

```
1:    | classDescr ivAndVal aClass classDef directedRel |
2:    classDescr := aModelDesc classNamed: aSymbol.
3:    directedRel := self
4:            relationForWhichBackPointerIsRequiredFor: aSymbol
5:            in: aModelDesc.
6:    ivAndVal := self
7:            ivsAndValuesForAttributes: classDescr attributes
8:            andRelations: directedRel.
9:    classDef := self
10:           computeClassDefinition: classDescr
11:           fromDescriptions: classDescr attributes , directedRel.
12:   Compiler evaluate: classDef.
13:   aClass := self generatedClass:
14:           (self smalltalkClassNameFromClassName: classDescr name).
15:   self compileInitializeMethodWith: ivAndVal in: aClass.
16:   self compileAccessorsForAttributes: classDescr attributes in: aClass.
17:   self compileMethodsForNaryAttributes: classDescr attributes in: aClass.
18:   self compileAccessorsForRelations: directedRel in: aClass[1].
19:   self compileMethodsForNaryRelations: directedRel in: aClass[2].
20:   self compileMOFDescriptionFor: aClass with: classDescr.
21:   self compileSuperNewFor: aClass.
22:   self compileOptimizeMethod: aModelDesc in: aSymbol
```

What we are doing here in the beginning is mostly trivial. The more interesting parts of this code (starting on line 9) will be explained in the following sections. First we get the class description itself. This is done by asking the meta-model description by the name that we received by calling this method.
In order to get all attributes of a class we have to resolve all attributes that the class defines by itself as well as all attributes that the class receives because of associations to other classes. To get the attributes of the class itself is fairly

[1]Please refer to the method compileAccessorsForAttributes: in:
[2]Please refer to the method compileMethodsForNaryAttributes: in:

simple, as it is done at the end of line 7 or line 11 for instance. To get the attributes the class gets from being involved in relations, we need to know in which relations the current class is involved (line 3 to 5). The code for this operation is shown in the Appendix A.
It is only good programming practice to always initialize instance attributes whenever a new instance of a class is created. So in the next step, on the lines 6 to 8, we associate each attribute which its initial value. This is stored for later computation and compilation. The initial value can be set or a default one will be chosen, in our case nil will be used. For attributes which have a multiplicity of more than one, a collection, like an OrderedCollection, will be stored as the initial value.

3.5 Class Definition Generation

computeClassDefinition: aInOutClassDescription
fromDescriptions: ivsAndValues

```
1:   | className stream stClassName |
2:   className := self classNameFromDescription: aInOutClassDescription.
3:   stClassName := self smalltalkClassNameFromClassName: className.
4:   stream := WriteStream on: (String new: 30).
5:   stream nextPutAll: self namespaceName.
6:   stream nextPutAll: ' defineClass: #', stClassName ; cr.
7:   stream tab ; nextPutAll: 'superclass: #{', self namespaceName, '.',
8:       self rootOfEntityHierachyClassName, '}' ; cr.
9:   stream tab; nextPutAll: 'indexedType: #none' ; cr.
10:  stream tab; nextPutAll: 'private: false' ; cr.
11:  stream tab; nextPutAll: 'instanceVariableNames: '; nextPut: $'.
12:  ivsAndValues do: [:ele |
13:      stream nextPutAll: ele name asString; nextPut: Character space].
14:  stream nextPut: $'; cr.
15:  stream tab; nextPutAll: 'classInstanceVariableNames: ""'; cr.
16:  stream tab; nextPutAll: 'imports: ""'; cr.
17:  stream tab; nextPutAll: 'category: '; nextPut: $';
18:      nextPutAll: modelDescription name asString; nextPut: $'.
19:  ^stream contents
```

As you can see, generating the definition code for a class looks quite difficult, but it is fairly simple actually. First we extract the name out of the class description and compute the smalltalk name with it. All the classes we generate are defined in the Moose namespace, including the class we already generated as our superclass. The name of this class is appended next to our description. Probably the most interesting part of the code starts on line 12. Here all the attributes for the instances are added to the class description. The important thing to notice here is that the collection of attributes (ivsAndValues) does not only contain all the attributes that the class defined by itself. It also includes all the attributes that have to be generated in order to store associative objects. (This can be seen in the previous Section 3.4 on page 7 in the code extract on line 11) Because we want to put our class in the environment we already generated for the class, we must add the name of our environment as the category to the class description. With the whole description generated, we can now pass this description to the compiler and let our class come to life. When our class has come to existance we can go on to do some more generation with it.

3.6 Initialize Generation

computeInitializeMethod: ivsAndValues

```
| stream |
stream := WriteStream on: (String new: 60).
self writeInitializeSelectorAndSuperCall: stream.
ivsAndValues do: [:each |
    stream tab;
    nextPutAll: each key;
    nextPutAll: ' := ';
    nextPutAll: each value asString;
    nextPutAll: '.'; cr].
^stream contents
```

As we already computed all the attributes and their initial values, it is now fairly simple to compute an initialize method. Because we want to call the initialize method in the superclass we first need to add the line of code that does just that. After that we can now add the code to initialize our attributes. To do so, we iterate over all attributes and one by one we add the name of the attribute and the initial value for the attribute.

3.7 Attribute Methods Generation

This section will cover the lines 16 to 19 of the code displayed in Section 3.4 on page 7. For each attribute that has been added to the class a number of methods is added as well. What kind of methods are being added depends purely on the multiplicity of the attributes. Here is an overview of the methods with all the multiplicities that are possible:

Multiplicity is zero

- Attributes that have a multiplicity of zero have already been ignored and nothing will be generated.

Multiplicity is one

compileAccessorFor: attributeDescription in: aClass

```
| iv |
iv := attributeDescription name.
self compile: aClass
    classified: 'accessing'
    source: (self computeGetterMethod: iv).
attributeDescription isToN
    ifTrue: [ self compile: aClass
                classified: 'private'
                source: (self computeSetterMethod: iv)]
    ifFalse: [ self compile: aClass
                classified: 'accessing'
                source: (self computeSetterMethod: iv)]
```

- A set method to set the value of the attribute is generated. This method can be used in the public interface of the class.
- A get method to get the value of the attribute is generated which can always be used in the public interface of the class.

Multiplicity is unlimited

compileIteratingMethodFor: iv in: aClass

```
self compile: aClass
    classified: 'aggregation iterating'
    source: (self computeDoMethod: iv).
self compile: aClass
    classified: 'aggregation iterating'
    source: (self computeCollectMethod: iv).
self compile: aClass
    classified: 'aggregation iterating'
    source: (self computeSelectMethod: iv).
self compile: aClass
    classified: 'aggregation iterating'
    source: (self computeRejectMethod: iv).
self compile: aClass
    classified: 'testing'
    source: (self computeIsEmptyMethod: iv)
```

compileAddMethodForAttribute: iv in: aClass

```
self compile: aClass
    classified: 'adding'
    source: (self computeAddMethodForAttribute: iv)
```

- A set method to set a collection of values is generated, but it is not intended to be used in the public interface of the class.
- A get method is generated to get the collection of values that are set. This method is always in the public interface of the class.
- A testing method is generated to check if the current collection of values is empty.
- A method to add a value to the current collection of values is generated.
- A method to iterate over the current collection and execute statements on each element is generated.

- A method to iterate over the current collection, execute statements on each element and collect the results of these statements is generated.
- Two methods to iterate over the current collection and collect or reject elements that fulfill a certain requirement are generated.

One must keep in mind that the definitions of the attributes have already been generated when the class definition has been generated. So this part of the generation will not be redone.

3.8 MOF Description Generation

computeMOFDescriptionFor: aClassDescription

```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: 'initializeMofDescription'; cr; cr; tab;
        nextPutAll: '^(MOFExtendedClass new)'; cr; tab; tab;
        nextPutAll: 'describeModelClass;'; cr; tab; tab;
        nextPutAll: 'name: #' , aClassDescription name, ';'; cr; tab; tab.
aClassDescription attributes do: [:each |
        stream nextPutAll: (self computeAttributeDescriptorFor: each)].
stream nextPutAll: 'yourself'.
^stream contents
```

computeAttributeDescriptorFor: anAttribute

```
| stream initialValue attributeType |
stream := WriteStream on: (String new: 60).
(anAttribute isMultiValue)
        ifTrue: [attributeType := 'MOFExtendedMultipleValueAttribute']
        ifFalse: [attributeType := 'MOFExtendedAttribute'].
stream nextPutAll: 'addAttribute: ((' , attributeType , ' new)';
        cr; tab; tab; tab;
        nextPutAll: 'name: #' , anAttribute name , ';';
        cr; tab; tab; tab;
        nextPutAll: 'loadMethod: #' , anAttribute loadMethod , ';';
        cr; tab; tab; tab.
stream nextPutAll: anAttribute type name asString , ';';
        cr; tab; tab; tab;
        nextPutAll: (constantsMap at: anAttribute multiplicity) , ';';
        cr; tab; tab; tab.
initialValue := anAttribute value.
((initialValue at: 1) = $')
        ifFalse: [initialValue := '''' , initialValue , '''']
        ifTrue: [initialValue := '''''' , initialValue , ''''''].
stream nextPutAll: 'initialValue: ' , initialValue , ';';
        cr; tab; tab; tab;
        nextPutAll: 'yourself);';
        cr; tab; tab.
^stream contents
```

Generating a MOF description for a class on the class side is meant for one reason: to import model data. Although it is quite handy to have a MOF description on the class side of the class in question, there is one problem with it: you cannot store associative attributes there and that is a handicap for storing models. So the generation of a MOF description based on the original MOF description will almost certainly vanish in near future. About the generation itself, there is not much to say because it is mainly a copying of the one we receive as a parameter.

3.9 Symbolic Link Resolver (Optimize) Generation

computeOptimizeMethod: aModelDescr for: aClassSymbol

```
| stream otherParticipant pointsFrom |
stream := ReadWriteStream on: (String new: 60).
stream nextPutAll: 'optimize: anImporter'; cr; cr; tab.
stream nextPutAll: '| anEntity aCollection |'; cr; tab.
(aModelDescr associationsTo: aClassSymbol) do: [:each |
        otherParticipant := each otherEntity: aClassSymbol.
        pointsFrom :=each directedRelationPointingTo: otherParticipant.
        (pointsFrom isNil not)
            ifTrue: [self computeSingleOptimize: each
                        on: stream
                        for: aClassSymbol.]].
^stream contents
```

computeSingleOptimize: aRelation on: stream for: aClass

```
| pointsTo otherParticipant pointsFrom |
otherParticipant := aRelation otherEntity: aClass.
pointsTo := aRelation directedRelationPointingTo: aClass.
pointsFrom := aRelation directedRelationPointingTo: otherParticipant.
(pointsFrom isToN)
    ifTrue: [stream nextPutAll: 'aCollection := OrderedCollection new.';
                cr; tab;
                nextPutAll: 'self ' , pointsFrom name asString ,
                    ' do: [:each | anEntity := anImporter currentModel
                    classWithName: each.';
                cr; tab; tab; tab; tab; tab; tab.
                self computeAddInOptimize: pointsTo on: stream.
                (pointsTo isNil not)
                    ifTrue: [stream tab; tab; tab; tab; tab].
                stream nextPutAll: 'aCollection add: anEntity].'; cr; tab;
                nextPutAll: 'self ' , pointsFrom name asString ,
                    ': aCollection.'; cr; tab]
    ifFalse: [stream nextPutAll: 'anEntity := anImporter currentModel
                    classWithName: self ', pointsFrom name asString , '.';
                cr; tab.
                self computeAddInOptimize: pointsTo on: stream.
                stream nextPutAll: 'self ', pointsFrom name , ': anEntity.';
                cr; tab]
```

computeAddInOptimize: aRelationDirection on: stream

```
(aRelationDirection isNil not)
    ifTrue: [stream nextPutAll: '(anEntity isNil not) ifTrue: [anEntity '.
            (aRelationDirection isToN)
                ifTrue: [stream nextPutAll: (
                            self computeNameOfAddMethod:
                                aRelationDirection name asString)]
                ifFalse: [stream nextPutAll: (
                            self computeNameOfSetterMethod:
                                aRelationDirection name asString)].
                            stream nextPutAll: ' self].'; cr; tab].
```

As this project was mainly about saving a model in a way that it could be reloaded, this is the most important part of the project. The goal was to store a model, reload it and you would not know the difference between the original model and and reloaded one. To store an interrelated model on disk has one potential problem: How do you store related objects?
One could think that just saving the model to the disk with all the objects in their original place might be a good solution. But that generally poses a number of problems because of containment.
One possible solution is to give each object in your model a unique identifier. When saving the model to disk, in every place an object occurs in a related fashion, it is replaced by its identifier. This is done for each object in the model. After that step you just have objects that are related to any other object only by symbolic links and not by themselves. In this state the model is saved to disk. Now on reloading the model, some objects should be in association with other objects. But it only has a symbolic link to any object in the model. To be able to resolve this link a method is being generated that replaces these symbolic links with the real object when reloading the model.
As you can see in the code, something is done for every association that is pointing to the class we are currently generating. If the direction can be navigated in the reverse way, lines of code to resolve the symbolic links for that association are then being generated. What happens then depends on the multiplicity of the relation direction from the class currently being generated.

Multiplicity is limited

The content of the attribute that is named after the association end we are currently pointing from, serves as name to get the object that is named accordingly to that name. One has to keep in mind, that this name is no ordinary name, but the symbolic link to an other object. The add code to store the object we are now currently optimizing in the resolved object is being generated now. The add code depends on the multiplicity of the association end we are pointing to. If the multiplicity is unlimited, a true add line will be appended. If it was limited, a simple set line will be appended. This only happens, if the object we have been looking for exists. As a last thing to do, we set the found object as the value for the attribute named after the association direction.

Multiplicity is unlimited

> First a new collection is generated to later hold the objects found in the system after the symbolic link to them has been resolved. Then we access all the symbolic links in the attribute that is named after the association end we are pointing from and iterate over them. For each link, we first get the object that belongs to that link. As with the limited multiplicity a line to add the object currently being optimized to the resolved object is being generated by the same rules. Then the resolved object is put in the collection and the next link is resolved. When all the links have been resolved, the collection with the resolved objects is set on the object we are currently optimizing.

With all these things generated for the current class, the generation ends and the next class will be generated until the whole meta-model is built. Now the model data can be imported into Moose and the work with the model may begin.

Chapter 4

Case Study

To validate our project we made a MOF description of an example meta-model of a simple document handling system with authors, documents, sections and tables of content. A simple UML-diagram of this demonstration meta-model can be seen in Figure 4.1. On the last pages of this chapter a small piece of the code generated from these descriptions can be found.

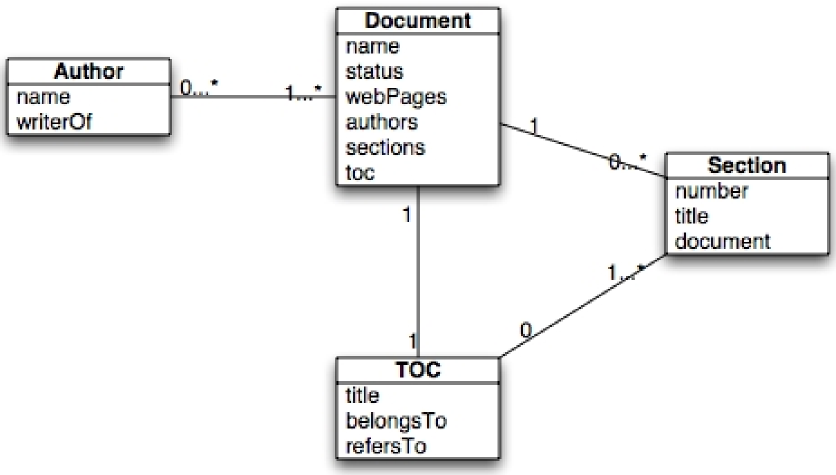


Figure 4.1: Demonstration meta-model of a simple document handling system

The complete set of MOF descriptions necessary to generate the demonstration meta-model can be seen in the Appendix B on page 26.
The procedure to get a MOF description of a system is really straight forward and will be shown on the following pages. The author class will serve as an example.

4.1 MOFClass Description

A class can have a variety of attributes. As you can see, an author object only has two attributes: name and writerOf. To write a class description of an class, one must only take the attributes of the class itself into account, not those that are inherited from a association with another class. In case of the author class this would just be the attribute name. The attribute writerOf is the collection of the documents an author has written, so that clearly is a associational attribute. This already is all the necessary information we need in order to write the description of a class in MOF. The general structure of such a description is the following.

MOFClass Description

```
^MOFExtendedClass new
    [describe(Model/Implementation)Class;]
    name: #(some class name);
    [addAttribute: ((MOFExtendedAttribute new)
                        name: #(some attribute name);
                        [loadMethod: #(some method name):;]
                        [someType;]
                        [multiplicity;]
                        initialValue: 'some initial value';
                        yourself);]
    [addAttribute: ((MOFExtendedAttribute...
    ...    ]
    yourself
```

There are some things you need to know about a class description and should be kept in mind when writing a description.

- The difference between a MOFClass and a MOFExtendedClass is that for a MOFExtendedClass it may be specified if it is model or an implementation class. For a MOFClass it is being assumed that it is a model class. If not specified it is being assumed that the class is a model class.
- A class does not necessarily need attributes of itself. But it may have as many as are necessary.
- Specifying a load method for an attribute is not absolutely necessary, although it is good practice to do so. The default value is #default:.
- Specifying the type of an attribute is not necessary as well. The default type of an attribute is identifier.[1]
- Specifying the multiplicity of an attribute may be skipped. The default value is that an attribute has a multiplicity of oneToOne.[2]

[1]Valid types are boolean, identifier, index, name, qualifier and string

[2]Valid multiplicities are zeroToZero, zeroToOne, zeroToN, oneToOne and oneToN

- The string quotation marks for the initial value are essential! If you want to have a string, e.g. the string 'test', as an initial value it must be written "'test"'. Or the initial value nil would be passed as 'nil'.

The MOF description of the author class can be found in the Appendix B on page 26.

4.2 MOFAssociation Description

A class may participate in a number of associations with other classes. Usually for a association, each participating class will receive an attribute to store related instances of the other participating entitiy of the association. In the case of the author class the writerOf attribute will be generated because of the association to the document class. But as you can see in Figure 4.1 the section class does not get an attribute of its association to the toc class. That is because the multiplicity of the association in the direction from the section class to the toc class is zero. So before you start debugging a not generated associational attribute, always have a look the multiplicity of the association. Now we can take a closer look at an association description:

MOFAssociation Description

```
^MOFAssociation new
    firstAssociationEnd: ((MOFAssociationEnd new)
                              endName: #class one name;
                              name: #association name to class two;
                              multiplicity;
                              yourself);
    secondAssociationEnd: ((MOFAssociationEnd new)
                              endName: #class two name;
                              [name: #association name to class one;]
                              multiplicity;
                              yourself);
    yourself
```

Again, two remarks have to be made.

- The name of the association to the first class may only be omitted if the multiplicity of the direction is zeroToZero
- The valid mulitplicities of a association direction are the same ones as with an ordinary attribute[3]

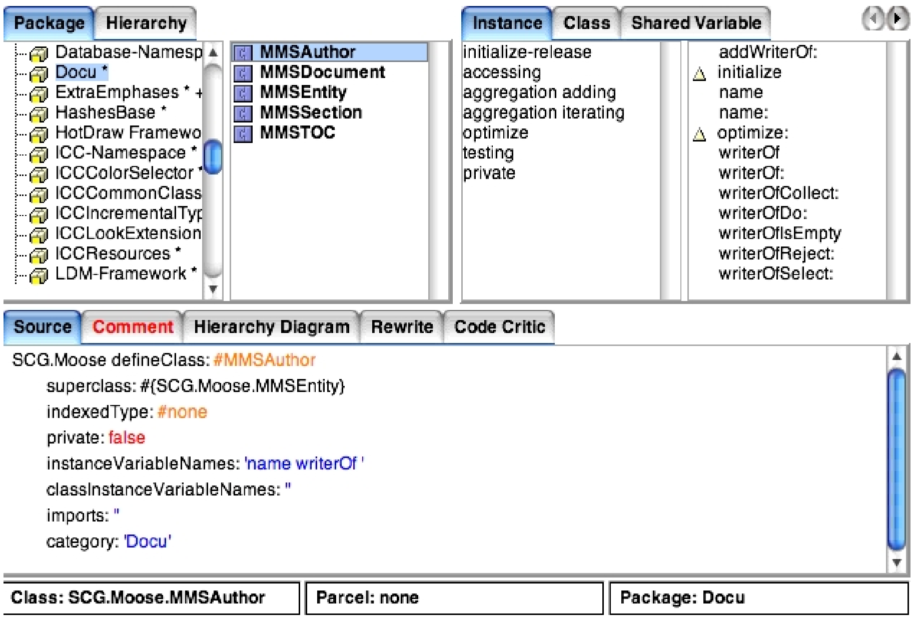
The MOF association description of the author class to the document class can be found in the Appendix B on page 28

4.3 Generated Meta-Model or Code

In order to be able to generate the code for the demonstration meta-model all class and association descriptions have to be written. In the next step, all

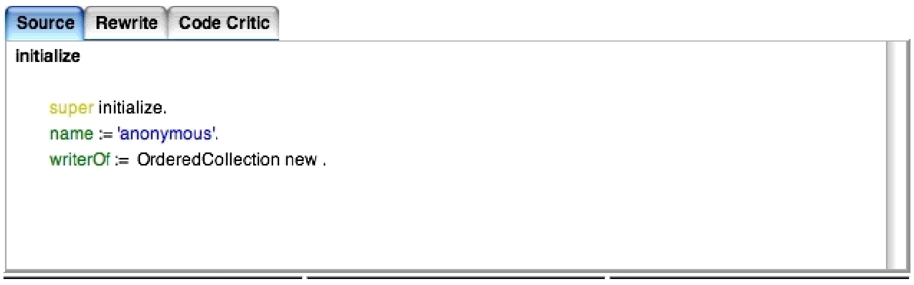
[3]Valid multiplicities are zeroToZero, zeroToOne, zeroToN, oneToOne and oneToN

class and association descriptions have to be collected in a description collector, a MOFExtendedPackage in this case. Every model needs a name, which is being used to name the parcel holding the classes that will be generated. This MOFExtendedPackage can then be passed to the DomainModelGenerator which will then generate the meta-model for you. Screenshots of the generated code of the auther class can be found in Figure 4.2 and in Figure 4.3.



```
SCG.Moose defineClass: #MMSAuthor
    superclass: #{SCG.Moose.MMSEntity}
    indexedType: #none
    private: false
    instanceVariableNames: 'name writerOf '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Docu'
```

Figure 4.2: Overview of the generated class author



```
initialize

    super initialize.
    name := 'anonymous'.
    writerOf := OrderedCollection new .
```

Figure 4.3: The generated author initialize method

Chapter 5

Conclusions

With the mechanisms that have been put in place, a true communication between tools and Moose is possible. Not only is it possible to generate a meta-model based on an implementation independant description of the meta-model, one can actually load, change and save a model with given data once the meta-model has been generated. Like this Moose can load a software model that another rengineering tool has created and vice versa.
As we saw in the preceding chapter, MOF descriptions are quite easy to read and understand and they generally do not take long to write. So if needed, to write a meta-model description in MOF by hand is not all to complicated. So even computer-human communication is possible, not only computer-to-computer communication.
Before such communication was possible, the effort that would have been needed to set up a meta-model like the one in the case study (see Figure 4.1 on page 15) and import the data of the model would have been managable. But when thinking about bigger systems the effort would have been very pricy if not unmanagable. An example for such a meta-model is shown in Figure 5.1 (Bugzilla) on page 20.
A possible disadvantage has to do with the code that is generated when building the meta-model. It is possible that unnecessary or unneeded methods are being generated. For instance not everyone may need all the different iterating methods.
Over all we consider it a very handy tool that really makes work with reengineering tools and Moose comfortable and fun.

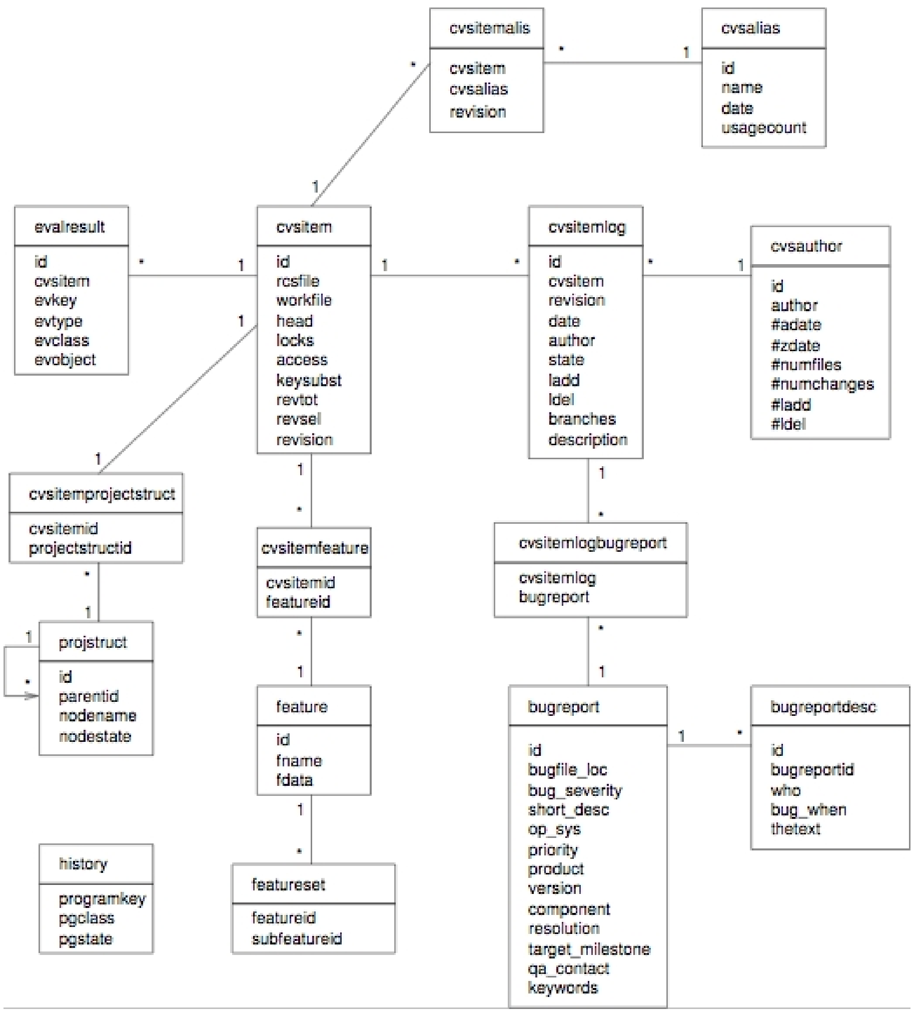


Figure 5.1: The bugzilla meta-model

Appendix A

Method Implementations

The methods of the MOFDomainModelGenerator not being displayed in the main document (listed in lexographical order)

```
additionCodeForNaryAttributes
   ^' add: anEntity'

classNameFromDescription: aDescription
   ^aDescription name asSymbol

compile: aClass classified: aString source: source
   aClass compile: source classified: aString notifying: nil

compileAccessorsForAttributes: attributesDescr in: aClass
   attributesDescr do: [ :each |
            self compileAccessorFor: each in: aClass ]

compileInitializeMethodWith: ivsAndValues in: aClass
   self compile: aClass
      classified: 'initialize-release'
      source: (self computeInitializeMethod: ivsAndValues)

compileMethodsForNaryAttributes: attributes in: aClass
   |naryAttr|
   naryAttr := attributes select: [:each | each isToN].
   naryAttr := naryAttr collect: [:each | each name].
   naryAttr do: [:each |
            self compileIteratingMethodFor: each in: aClass.
            self compileAddMethodForAttribute: each in: aClass ].

compileMOFDescriptionFor: aClass with: aClassDescription
   self compile: aClass class
      classified: 'mof'
      source: (self computeMOFDescriptionFor: aClassDescription)
```

compileOptimizeMethod: aModelDescr in: aSymbol
```
self compile: (self generatedClass:
                (self smalltalkClassNameFromClassName: aSymbol))
    classified: 'optimize'
    source: (self computeOptimizeMethod: aModelDescr for: aSymbol)
```

compileRootEntity
```
| stclass |
Compiler evaluate: self computeClassRootEntityDefinition.
stclass := self class environment
        at: (self smalltalkClassNameFromClassName: 'Entity').
self compileSuperNewFor: stclass
```

compileSuperNewFor: aClass
```
self compile: aClass class
classified: 'instance creation'
source: (self computeSuperNew)
```

computeAddMethodForAttribute: iv
```
| aStream |
aStream := WriteStream on: (String new: 60).
aStream
        nextPutAll: 'add';
        nextPutAll: (self stringWithFirstUppercasedCharacter: iv asString);
        nextPutAll: ': anEntity'; cr; cr; tab.
aStream nextPutAll: iv; space.
aStream nextPutAll: self additionCodeForNaryAttributes.
ˆaStream contents
```

computeCollectMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; nextPutAll: 'Collect: aBlock' ; cr; cr ; tab;
        nextPutAll: 'ˆ'; nextPutAll: iv ; nextPutAll: ' collect: aBlock' .
ˆstream contents
```

computeDoMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; nextPutAll: 'Do: aBlock' ; cr; cr ; tab;
        nextPutAll: iv ; nextPutAll: ' do: aBlock' .
ˆstream contents
```

computeGetterMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; cr; cr ; tab ;nextPutAll: 'ˆ'; nextPutAll: iv.
ˆstream contents
```

computeIsEmptyMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; nextPutAll: 'IsEmpty' ; cr; cr ; tab;
        nextPutAll: '^'; nextPutAll: iv ; nextPutAll: ' isEmtpy' .
^stream contents
```

computeNameOfSetterMethod: iv
```
^iv, ':'
```

computeRejectMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; nextPutAll: 'Reject: aBlock' ; cr; cr ; tab;
        nextPutAll: '^'; nextPutAll: iv ; nextPutAll: ' reject: aBlock' .
^stream contents
```

computeSelectMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: iv ; nextPutAll: 'Select: aBlock' ; cr; cr ; tab;
        nextPutAll: '^'; nextPutAll: iv ; nextPutAll: ' select: aBlock' .
^stream contents
```

computeSetterMethod: iv
```
| stream |
stream := WriteStream on: (String new: 60).
stream
    nextPutAll: (self computeNameOfSetterMethod: iv);
    nextPutAll: ' anObject' ; cr; cr ; tab ;
    nextPutAll: iv ;
    nextPutAll: ' := anObject'.
^stream contents
```

computeSuperNew
```
| stream |
stream := WriteStream on: (String new: 60).
stream nextPutAll: 'new'; cr; cr; tab;
nextPutAll: '^super new initialize'.
^stream contents
```

containerCodeForNaryAttributes
```
^' OrderedCollection new '
```

```
generatedClass: aClassName
    | class |
    class := self class environment at: aClassName ifAbsent: [nil].
    class isNil
        ifTrue: [self error: 'the class ', aClassName ,
                                ' has not been successfully generated']
        ifFalse: [^class]

ivsAndValuesForAttributes: anAttributeDescription
    ^anAttributeDescription collect: [:each |
        each isToN
            ifTrue: [each name asString -> self containerCodeForNaryAttributes]
            ifFalse: [each name asString -> (each initialValueIfAbsent: ['nil'])]]

ivsAndValuesForAttributes: attributeDescriptions
    andRelations: directedRelations
    ^(self ivsAndValuesForAttributes: attributeDescriptions)
        , (self ivsAndValuesForRelations[1]: directedRelations)

metaModelClassPrefix
    ^'MMS'

namespaceName
    ^'SCG.Moose'

relationForWhichBackPointerIsRequiredFor: aClassSymbol
    in: aModelDescription
    | fromRels fromDirectedRels |
    fromRels := aModelDescription associationsFrom: aClassSymbol.
    fromDirectedRels := fromRels collect: [:each |
            each directedRelationFrom: aClassSymbol].
    ^fromDirectedRels reject: [:each |
            each isZero]

rootInMoose
    ^'AbstractEntity'

rootOfEntityHierachyClassName
    ^self metaModelClassPrefix , 'Entity'

smalltalkClassNameFromClassName: aStringOrASymbol
    ^(self metaModelClassPrefix , aStringOrASymbol) asSymbol
```

[1]For the code of the method ivsAndValuesForRelations please refer to the method ivsAndValuesForAttributes.

stringWithFirstUppercasedCharacter: aString

```
|stringCopy|
stringCopy := aString copy.
stringCopy at: 1 put: (aString at:1) asUppercase.
^stringCopy
```

writeInitializeSelectorAndSuperCall: stream

```
stream nextPutAll: 'initialize'; cr; cr; tab;
        nextPutAll: 'super initialize.'; cr
```

Appendix B

MOF Descriptions

The complete list of MOF Descriptions to be able to generate the demonstration model on page 15

MOFClass descriptions

Author

```
^MOFExtendedClass new
describeModelClass;
name: #Author;
addAttribute: ((MOFExtendedAttribute new)
                    name: #name;
                    loadMethod: #name:;
                    identifierType;
                    zeroToOne;
                    initialValue: ''anonymous'';
                    yourself);
yourself
```

Document

```
^MOFExtendedClass new
describeModelClass;
name: #Document;
addAttribute: ((MOFExtendedAttribute new)
                    name: #name;
                    loadMethod: #name:;
                    identifierType;
                    oneToOne;
                    initialValue: ''the bible'';
                    yourself);
addAttribute: ((MOFExtendedAttribute new)
                    name: #status;
                    loadMethod: #status;
                    booleanType;
                    zeroToOne;
                    initialValue: 'nil';
```

```
                        yourself);
addAttribute: ((MOFExtendedMultipleValueAttribute new)
                        name: #webPages;
                        loadMethod: #webPages;
                        identifierType;
                        zeroToN;
                        initialValue: 'OrderedCollection new';
                        yourself);
yourself
```

Section

```
^MOFExtendedClass new
describeModelClass;
name: #Section;
addAttribute: ((MOFExtendedAttribute new)
                        name: #number;
                        loadMethod: #number:;
                        indexType;
                        oneToOne;
                        initialValue: '0';
                        yourself);
addAttribute: ((MOFExtendedAttribute new)
                        name: #title;
                        loadMethod: #title:;
                        stringType;
                        zeroToOne;
                        initialValue: '' '';
                        yourself);
yourself
```

TOC

```
^MOFExtendedClass new
describeModelClass;
name: #TOC;
addAttribute: ((MOFExtendedAttribute new)
                        name: #title;
                        loadMethod: #title:;
                        stringType;
                        zeroToOne;
                        initialValue: '' '';
                        yourself);
yourself
```

MOFAssociation descriptions

Author Document Association

```
^MOFAssociation new
firstAssociationEnd: ((MOFAssociationEnd new)
                        endName: #Author;
                        name: #writerOf;
                        oneToN;
                        yourself);
secondAssociationEnd: ((MOFAssociationEnd new)
                        endName: #Document;
                        name: #authors;
                        zeroToN;
                        yourself);
yourself
```

Document Section Association

```
^MOFAssociation new
firstAssociationEnd: ((MOFAssociationEnd new)
                        endName: #Section;
                        name: #document;
                        oneToOne;
                        yourself);
secondAssociationEnd: ((MOFAssociationEnd new)
                        endName: #Document;
                        name: #sections;
                        zeroToN;
                        yourself);
yourself
```

Document TOC Association

```
^MOFAssociation new
firstAssociationEnd: ((MOFAssociationEnd new)
                        endName: #TOC;
                        name: #belongsTo;
                        oneToOne;
                        yourself);
secondAssociationEnd: ((MOFAssociationEnd new)
                        endName: #Document;
                        name: #toc;
                        oneToOne;
                        yourself);
yourself
```

TOC Section Association

```
^MOFAssociation new
firstAssociationEnd: ((MOFAssociationEnd new)
                          endName: #TOC;
                          name: #refersTo;
                          oneToN;
                          yourself);
secondAssociationEnd: ((MOFAssociationEnd new)
                          endName: #Section;
                          zeroToZero;
                          yourself);
yourself
```