Template Programming for Model-Driven Code Generation

Author: Ghica van Emde Boas
e-mail: emdeboas@bronstee.com

Contents

Abstract

The purpose of this paper is to describe the state of my (not “the”) art of template programming.

Template programming for model-driven code generation needs to be placed in a context, therefore we describe a small model-driven generative software development project.

We show how to implement a simple meta-model into a modeling tool and how to define an application model using this tool. Then we show how templates can be developed to generate meaningful code for the application. We look at the two most popular template languages: Velocity and JET. In the spirit of contributing best pratices, we describe at the end of this paper some issues we have encountered in practice while developing code generation templates.

Introduction

For the **Generative Model Transformer** (GMT)[1] project we are developing a prototype that should show the development process of a typical Model-Driven Software Development project in a practical setting.

Here we are describing a *simplified* version of this prototype, using *FUUT-je*[2]. We have also implemented this prototype using EMF[3], the Eclipse Modeling Facility. The principles are very similar and we will make some remarks on the use of EMF at the end of the paper.

Background

Many template languages are either proprietary or the result of some ad-hoc implementation project. Providing well-defined syntax and semantics for the type of language we describe here, has not been done as far as we know. This could be an interesting research project.

The closest we can think of as a fomal definition for template languages is the definition of XSLT (Extensible Stylesheet Language for Transformations)[4]. XSLT is too verbose to be an effective language for model-driven code generation, but for preforming XML transformations it has been proven to be very popular. An early attempt to define a model driven template language as well defined XML, the built-in the template language of FUUT-je, can be found in[5].

The two most popular template languages in the Java environment, JET and Velocity, owe their power to their resemblance to *form-letters*. The template is as close as possible to the output of Java source code, HTML or text.

Other methods to generate source code include using rules, writing programs to write programs, and using transfomations such as XSLT. We have seen both a rule based code generator and a complex-program based code generator for the very large IBM SanFrancisco Java framework (now succeeded by the various Websphere Components frameworks). Both approaches resulted in overly complex and difficult to use pieces of software. We believe this is caused by designing software with a complex model structure in mind instead of the flat, sequential structure of a Java program that can easily be shown in a template.

Note that we are talking here about **model-to-code** or flat text generation, not model-model transformation. In this area, graph rewriting, rules etc. may be more effective. Related subjects, such as QVT[6] are therefore outside the scope of this paper.

Model-Driven Software development

The reasons for doing Model-Driven Software Development or doing development in the MDA style can be found at our http://www.mdsd.info site, or at the MDA site of the OMG: http://www.omg.org/mda. Here we concern ourselves with the technicalities of building modeling tools and how to interact with templates that should provide us with better and more easy to use code generation facilities than currently available.

Programming in today's mainstream programming languages such as Java is theoretically still possible using a **notepad** editor and command line operations for compiling and executing code. In practice, most programmers will use a combination of programming IDE's, wizards and code generation tools to make his/her development more productive. The next step in abstraction, using model-driven techniques, is not widespread yet.

Why would you do dull and repetitive work when you could be innovative and produce better code at the same time by using the *model-driven tools and methods* that are currently available? Effective model-driven development cannot be done however, without some amount of custom template programming to provide generated code in the *style, using the standards, and for the enveronment or platform* of your company or organisation.

Template programming is still an art in its infancy, the template languages are not mature either. Still, we hope to show that you do not need a 20 person project to be able to benefit from model-driven code generation techniques and that you can do useful template programming with minimal effort and maximal result.

Model-Driven, Generative Development

According to the methodology put forward by the Jorn Bettin at MDSD.info[7], the development process for building an application using model-driven and generative techniques, will roughly unfold in the following steps:

1. Develop a *meta-model* for the environment where the application will run[8].
2. Develop a modeling tool from this meta-model.
3. Develop a *model* of the application.
4. Develop code generation *templates* to generate an application from the model in the context of the meta-model.
5. Generate the code!
6. Add *manual code* at encapsulated spots. For the time being, this will remain necessary.
7. Iterate.

Step 7 is an important step in this process. A continuous involvement of domain experts, architects and technical experts is needed to make model-driven development an agile process that can compete with any other agile method in productivity and efficiency.

As you can see, you need to develop two applications instead of one. The first application is a modeling tool that can be used to develop the application model. Over time it is expected that we will have tools to generate these modeling tools and/or that ready-built tools for many domains will become available.

The Example

The Meta-model

For our experiment we are using an extremely simple meta-model. Its purpose is to allow us to generate Java code from an application model. Fig 1. shows what the meta-model looks like.

We accept criticism for this meta-model. It is too simple for a real life situation, and it could be optimized. For example, useful attributes like *visibility* are missing and we could put the *name* attribute that each class seems to have in a common super-class.

This simple model serves its purpose well enough to explain our example without adding too much complexity and therefore we think it is useful. The model could be easily expanded into a usable model for Java code generation..

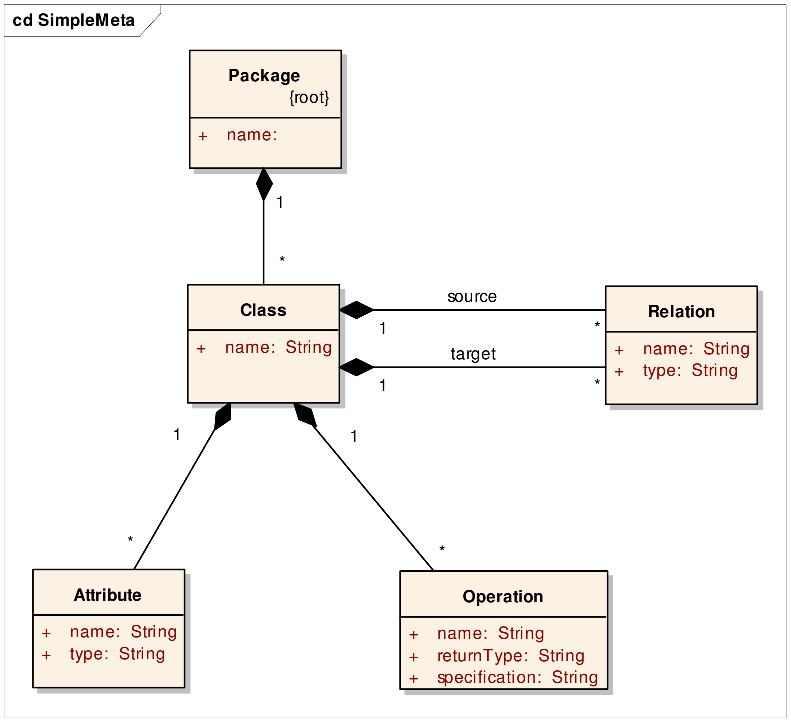


Fig. 1 – Very simple meta-model for the Java language.

The Application

As an example **application**, we chose the ubiquitous *GuestBook.* Here we only show how to generate some POJO's (Plain Old Java Objects)[9] that could be used as part of such an application, in a real setting you would probably like to implement this as a web-application, of which these POJO's could be part.

The model for the GuestBook application is shown in figure 2.

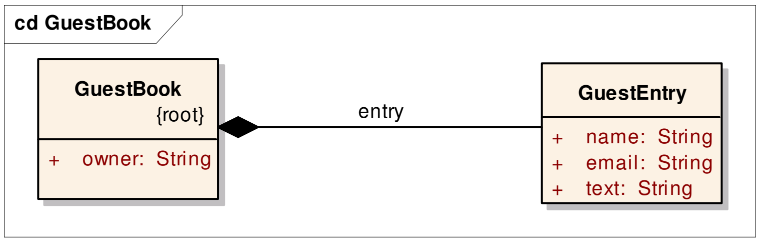


Fig. 2 – The GuestBook model in UML

As a *side note* we should mention that actually we would like the GuestBook model to be **independent** of the meta-model we just defined, because we would like to be able to also implement an application with the same functionality in other environments, *PHP* or *.net* for example. In the original GMT philosophy, there would be two models in some abstract meta-modeling language (MOF? ECore? UML?),[10] one for the platform and one for the application. The two would be combined using a mapping into a platform dependent model (PSM) as the Y-shape picture in figure 3 (taken from the GMT documentation) suggests.

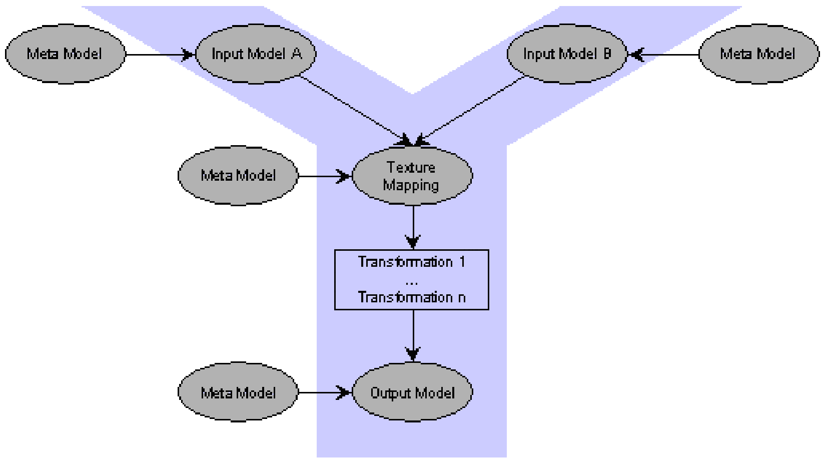


Fig 3. - The basic pattern for model transformations

Our technology is not good enough yet to do model merging in practice, therefore we are performing the step of mapping the application model to the meta-model implicitly and manually.

Implementation of the Meta-Model into a Modeling Tool

After the meta-model is defined, we need to implement a tool that allows the user to define models for an application in the context of the meta-model. As described, in our example the meta-model is the *simple Java model* as shown in figure 1.

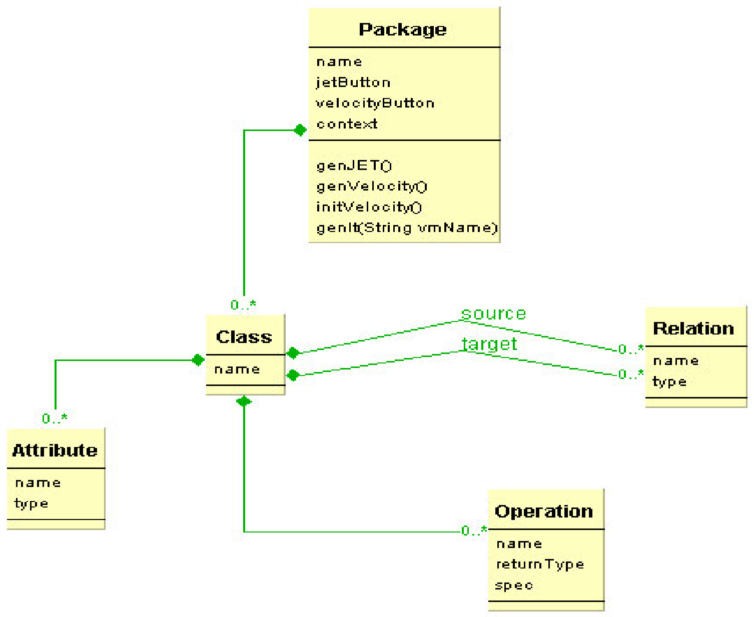


Fig. 4 – the FUUT-je model for the Simple Java meta-model

First we must define the meta-model as a *model* in FUUT-je. You can see it in figure 4. It looks the same as in figure 1, with some extra attributes and methods. The extra attributes, *jetButton and velocityButton* in the **Package** class specify, as their names suggest, buttons the user can press on the generated GUI to start the code generation. Because we would like to contrast the two template languages, JET and Velocity, there are two buttons. The *context* attribute is needed to keep the *Velocity* context information.

If you need help developing this model using FUUT-je, look at the FUUT-je tutorial at the GMT site, http://www.eclipse.org/gmt. The code for the example will also be available here.

We can generate code from the *Simple Java meta-model-model* using the *SwingSet* set of FUUT-je templates. This will produce a Java Swing application with a ready built GUI that we can use to define application models.

At this stage it is not so important to look into the details of the application generation in FUUT-je. To summarize, FUUT-je has its own template language, used for the SwingSet, that we will not describe here. It is also possible to use Velocity with FUUT-je. There exist a set of Velocity templates intended to generate *PHP* code from FUUT-je models.

It is important however to look back at the section Model-Driven, Generative Development, and see that we have completed step 1 and 2.

The result from the code generation step and some tuning is a **modeling tool** that we can use to perform step 3 of our development process: Develop a *model* of the application. See figures 5 for screenshots of the modeling tool. One of the windows not show has buttons to start the code generation.

Before we describe how the application data is entered into the modeling tool, we will complete the tool itself by adding *code generation facilities*. The development of actual templates will be done after the development of the application model, as suggested in our stepwise development process. In practice these will be parallel activities of course.

Calling Velocity to Start Code Generation

Setting up the model tool we just produced for code generation using Velocity templates is very simple. We need to place the Velocity jar in the *classpath* and we need a few lines of code to initialize the Velocity engine (not shown here). Before calling the generation engine, the Velocity context is filled with some data. The context is essentially a hashmap that can be referenced in a template.

```
public void genVelocity() {
  System.out.println("gen Velocity button pressed!");
        Vector classVec = this.getClasss();
        this.initVelocity(); // initialize the velocity template engine
        FtVelocityUtil vUtil = new FtVelocityUtil();
        for (int i=0;i<classVec.size();i++) {
                ClassData data = (ClassData)classVec.elementAt(i);
                attContext.put("ft", vUtil);
                attContext.put("package", data.getParentPackage().getName());
                attContext.put("newDate", new Date());
                attContext.put("class", data);
                System.out.println("=== Velocity generated class === : " + data.getName() + "\n");
                String result = genIt("vtemplates/gendemo.vm");
                System.out.println(result);
                System.out.println("=== end of Velocity class === : " + data.getName() + "\n");
        }
}
```

The *genVelocity()* method is called as action after pressing the “Generate with Velocity” button. The *genIt()* method basically contains

```
        Velocity.mergeTemplate(vmName, "UTF-8", attContext, w);
```

surrounded by a try-catch block to catch exceptions.

There are three things to note about calling the Velocity engine:

1. The name of the template is an argument to the *mergeTemplate()* method. It is easy to modify the shown code in such a way that the name of the template can be chosen dynamically at run-time.
2. The template is just a text file that is interpreted at runtime. You can configure Velocity to leave the template loaded, as would be useful at deployment of Velocity as a web-service generating HTML. For development purposes, it is useful to keep the templates unloaded, because it allows you to change the template and re-run the code generation without re-starting the generation tool. This is a *very important* productivity enhancer while developing and debugging templates.
3. The Velocity engine integrates easily within any Java environment, within Eclipse or outside.

Calling JET to Start Code Generation

Including the JET engine into the modeling tool is quite a bit more involved. JET is part of EMF, therefore it is necessary to install the EMF plugins into Eclipse before you can use JET. Maybe it is theoretically possible to run a modeling tool using JET outside of Eclipse, it is certainly not practical. To set-up your Eclipse project for using JET, a number of mouse-juggling steps is required, see the JET tutorial[11] for advice.

In the modeling-tool, after the set-up is done, calling the JET engine is easy:

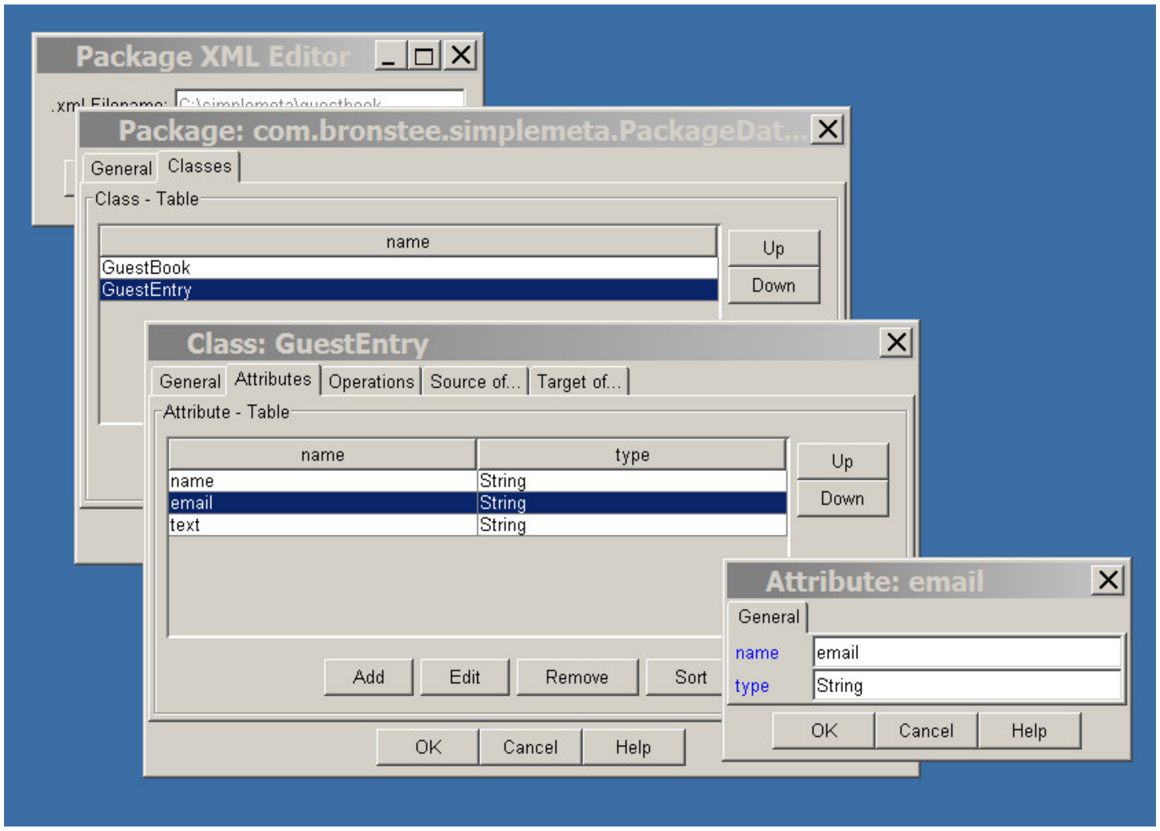
```
  public void genJET() {
    System.out.println("gen JET button pressed!");
            com.bronstee.demo.JavaDemoTemplate generateJava = new
com.bronstee.demo.JavaDemoTemplate();
            Vector classVec = this.getClasss();

            for (int i=0;i<classVec.size();i++) {
                        ClassData data = (ClassData)classVec.elementAt(i);
                        System.out.println("=== JET generated class === : " + data.getName() + "\n");
                        String result = generateJava.generate(data);
                        System.out.println(result);
                        System.out.println("=== end of JET class === : " + data.getName() + "\n");
            }
  }
```

The *genJET()* method is called after pressing the “Generate with JET” button in the modeling application. There are several things to note here:

1. With JET you do not call the engine with the name of the template, but with the name of the Java class generated from it. This makes it much more difficult to determine the template to be used dynamically at run time, it would need reflective techniques to do so.
2. Changing the template results in changing a Java class, and therefore the modeling tool needs to be restarted before the modified template can be used. *This is a serious drawback for development productivity.*
3. JET only integrates with Eclipse. This is a serious drawback for easy deployment of tools using JET.

Entering the Application Model Data

After some tuning of the GUI, the resulting application from generating code for the meta-model as defined in FUUT-je, looks as in figure 6. We can use the application editor to enter the model information for the GuestBook application model.



.xml Filename: C:\simplemeta\guestbook

Class - Table

name
GuestBook
GuestEntry

Attribute - Table

name	type
name	String
email	String
text	String

name: email
type: String

Fig 5. – The Simple Java meta-model editor

Of course it would have been much better and provide more user productivity features if we would have been able to generate a graphical editor which would allow us to enter the application model information, similar to an UML editor as shown in figure 2. Unfortunately we do not have the right toolset for this yet. Coding a graphical interface manually is outside the scope of this paper and is also not foreseen for the GMT prototype for which this work is a “prototype”.

Difficult as it may be, we have succeeded in entering the model information using the generated modeling tool editor. The tool has a facility to *serialize* the model data entered to an XML format. This will be the way to save and restore application models. For our GuestBook example we will get the following XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Package>
     <name>example</name>
   <Class>
        <name>GuestBook</name>
      <Source>
           <name>GuestEntry</name>
           <type>List</type>
        </Source>
      <Attribute>
           <name>owner</name>
           <type>String</type>
        </Attribute>
     </Class>
   <Class>
```

```
      <name>GuestEntry</name>
     <Target>
         <name>GuestBook</name>
         <type>GuestBook</type>
      </Target>
     <Attribute>
         <name>name</name>
         <type>String</type>
      </Attribute>
     <Attribute>
         <name>email</name>
         <type>String</type>
      </Attribute>
     <Attribute>
         <name>text</name>
         <type>String</type>
      </Attribute>
    </Class>
 </Package>
```

As you can see, thhe XML structure is not XMI, but very straightforward. Notice that there is no <Relation> tag as you might expect. Instead the relationship *names* <Source> and <Target> are used. This is a choice made for the FUUT-je implementation to make the generic parsing and un-marshalling of the XML into Java objects much easier.

Of course you could use *any XML editor* instead, to enter the model information, and then just use the generated modeling tool to open the XML file and generate the Java code.

Template Languages

In the previous sections we have developed our example *GuestBook* application model according to MDSD methodology. It is now time to look at the template languages we could use for code generation of the application and develop some actual templates as suggested in step 4 of our MDSD development process.

What is a Template Language?

Templates are as old as *form letters*. Any time you receive printed mail similar to this:

> Dear John Doe,
> We are happy to inform you …

there will be a template like this:

> Dear <<name>>,
> We are happy to inform you …

There will also be a program that keeps *<<name>> = 'John Doe'* expressions. At runtime all <<name>> occurrences are then replaced by 'John Doe'.

The popularity of template languages specifically designed to generate HTML is increasing at the same pace the use of the internet in dynamic ways is increasing. Examples of such languages are JSP and PHP. The same principles are now increaingly used to generate other text than HTML, such as Java code, XML configuration files, etc.

Two popular languages for code generation are **JET** that resembles JSP and **Velocity** that is more like PHP.

Requirements for a Template Language for Model-Driven Code Generation

Certainly the usual requirements for tlanguages in general apply: *ease of use* and *run time performance.* Above all, it should be easy to see in the template what the generated output will look like.

In addition, a template language that is suitable for model driven code generation should be aware of the structure of the meta model for a model to be able to easily navigate through the model.

There has been a discussion in the *eclipse.tools.emf* newsgroup at *news.eclipse.org* about JET versus Velocity as code generation template languages (Ed Willink pointed me to it)[12]. The implication seems to be that Velocity is more productive and that JET is more powerful. We are showing some templates and code for both, so you can decide for yourself.

Velocity

Velocity[13] is an Apache project. In the introduction it is said: “Velocity is a Java-based template engine. It permits anyone to use the simple yet powerful template language to reference objects defined in Java code.”

An Eclipse plugin to edit Velocity templates can be found at: http://sourceforge.net/projects/veloedit/.

The template language is indeed a very simple, but complete language:

- it has **#set()**, **#if(), #elseif(), #else()** and **#foreach()** constructs.
- A variable is anything that starts with a ‘$’.
- Variables are replaced with their values at generation time.

The premier quality of Velocity is its ability to navigate through Java structures. You can pass a list of Java objects and invoke methods on these objects. For example, if you had passed a **Person** object as *$person* to a Velocity template, you can get its name as follows:

```
Dear $person.getName(),
We are happy to inform you …
```

In the case of getters and setters you can also use a shorthand: `$person.Name` instead of `$person.getName()`.

For our *GuestBook* application, or more precise, for our modeling tool, we developed a simple template that can be used to generate Java skeleton code from the application model. Not just GuestBooks, but Java code for any model defined in the modeling tool.

This is the Velocity template:

```
/*
 * Created on $newDate
 * generated by a FUUT-je application using Velocity templates
 */

package $package;

public class $class.Name {
```

```
#foreach($att in $class.Attributes)
      protected $att.Type ${att.Name};
#end

#foreach($att in $class.Attributes)
      /**
       * @return Returns the $att.Name
       */
      #set( $uName = "${ft.capFirst($att.Name)}")
      public String get${uName}() {
            return $att.Name;
      }
      /**
       * @param $att.Name  The $att.Name to set.
       */
      public void set${uName}(String ${att.Name}) {
            this.${att.Name} = $att.Name;
      }

#end
}
```

Note about the Velocity language that it is **untyped**. This allows developers to write templates knowing not much about Java structures except its *structure*. No type casting etc. is necessary. Still, knowledge about the Java object that are passed is needed. For commonly used meta-models it may be desirable to encapsulate the meta-model structure as much as possible by developing utility classes.

When we execute the code shown in “Calling Velocity to Start Code Generation” code will be generated for a *GuestBook* class and a *GuestEntry* class. The output for the *GuestBook* class is:

```
/*
 * Created on Sun Aug 29 14:55:23 CEST 2004
 * generated by a FUUT-je application using Velocity templates
 */

package example;

public class GuestBook {

      protected String owner;

      /**
       * @return Returns the owner
       */
            public String getOwner() {
            return owner;
      }
      /**
       * @param owner  The owner to set.
       */
      public void setOwner(String owner) {
            this.owner = owner;
      }

}
```

The template code shown here is about as powerful as the best wizards that generate code from models, with the exeption of what EMF provides. With a little more effort it is possible to write templates that do much more and that can make every developer using these MDSD techniques very productive.

JET

The JET language resembles JSP very closely.

A JET editor plugin for Eclipse can be found at: http://sourceforge.net/projects/jet-editor.

Within the **<% ... %>** tags any Java code can be used, as for JSP. This makes JET more powerful as a template language than Velocity. It makes it also easier to create a mess in your templates. For our demo template that is not the case ☺, in fact it looks very similar to the Velocity template:

```
<%@ jet package="com.bronstee.demo" imports="java.util.*
com.bronstee.simplemeta.* org.eclipse.gmt.fuut.common.*"
class="JavaDemoTemplate" %>

/*
 * Created on <%=new Date()%>
 * generated by a FUUT-je application using JET templates
 */
<% ClassData cdata = (ClassData) argument;
      String className = cdata.getName();
      Vector atts = cdata.getAttributes();
      String packageName = cdata.getParentPackage().getName();
%>
package <%=packageName%>;

public class <%=className%> {

    <%
    for (Iterator i = atts.iterator(); i.hasNext(); ) {
            AttributeData elem = (AttributeData)i.next();
            String name = elem.getName();
            String type = elem.getType();
   %>
      public <%=type%> <%=name%>;
   <% } %>

    <%
    for (Iterator i = atts.iterator(); i.hasNext(); ) {
            AttributeData elem = (AttributeData)i.next();
            String name = elem.getName();
            String uName = FtUtil.capFirst(name,'u');
   %>

      /**
       * @return Returns the <%=name%>.
       */
      public String get<%=uName%>() {
            return <%=name%>;
      }
      /**
       * @param <%=name%> The <%=name%> to set.
       */
      public void set<%=uName%>(String <%=name%>) {
            this.<%=name%> = <%=name%>;
        }
    <% } %>
}
```

Note about the JET language that it is Java and therefore **typed**. It seems that writing JET templates requires intimate knowledge of the implementation of the underlying metamodel. This may be a drawback for developing user-friendly modeling tools that offer template faciliets based on JET.

The output for the *GuestBook* class is **exactly the same** as the output produced using the Velocity template and therefore it is not shown here.

Implementing the Example using EMF

EMF does not have a graphical front-end, except for the Omondo[14] plugin that is not yet available for Eclipse version 3 at the time of writing. The possibilities to import a model into EMF are to use either a Rational Rose[15] model, or an XML schema or annotated Java.

Implementation in EMF proceeds as follows:

- Define the matamodel, similar to the model in fig. 1, in Rose. We should also be able to use the Enterprise Architect model shown in fig. 1, exported to XSD.
- Create an EMF projects for the metamodel, and import the rose model. See the EMF[16] tutorial for information on how to do this.
- Generate the model code, the edit and the editor code.
- Create the guestbook model using the EMF generated editor for the simple tool model.
- Create Velocity templates.
- Implement a wrapper package that can load a meta-model instance, for example the *guestbook* model, and then calls the Velocity interpreter on a set of templates.

Using EMF seems quite a bit more involved than what is needed to expand a generated FUUT-je application as described above. We are still struggling with all the mouse juggling we have to do and we are tryingto describe this in a “cookbook” that can be used with GMT.

The true advantage of using EMF will be that the result is better integrated with Eclipse and that more tools will be able to interface with the EMF XMI formats.

Velocity Templates for EMF

It turns out that the templates to be used for for generating code from a specific metamodel are **exactly the same** for both the FUUT-je and EMF implementation of the metamodels, except for a truly minor difference in naming the access methods of collections that hold relationships between meta-classes.

Template Programming

In the example above we only scratched at the surface of the complexity of template programming. In this section we would like to add some notes of things you need to be aware of in practical template programming.

Template Utility Class

It will almost always be necessary to write a class that performs utility functions. In theory you would not need it for JET templates, in practice your templates will become unreadable very quickly.

A simple example that we encountered in our GuestBook application is the generation of getters and setters, where the naming convention suggests that *get/set* is followed by the attribute name with the *first character in uppercase*.

A certain need is also *validation* of names. In our Java models, the class, attribute, and method names should be valid Java identifiers and maybe also adhere to company standards, such as: attribute and method names start with lowercase, class names start with uppercase.

File Writing

In our simple example, the generated code was displayed as standard output. The template engines return a *string* as result of their work. In practice it is needed to write the generated code into a file. The knowledge which file should be written belongs naturally to the template. In the FUUT-je template language there is a special construct to open files of the right name, for Velocity and JET we have to use a trick to know the right filename. The trick is to define a field in the utility class that will be set by template programming. After the result string is returned by the generation engine, the calling code can read that field and write the complete result string to the specified file.

Conditional Template Structures

In our example we assumed that all attributes would have *public* visibility. The attributes are declared *protected*[17] with *public* getters and setters. In case your meta-model allows for *protected* or *private* visibility, the template that generates the attribute declarations and the getter/setter methods needs to check for each case and generate code accordingly. For these 3 visibilities, the template code becomes three times as long and it will be much less easy to see what the resulting code is.

Initial values or constraints for attributes can also result in a large amount of conditional template programming. Depending whether an initial value is supplied or not, you may need to declare a default value to avoid null pointer exceptions or compile errors.

Sometimes you can avoid conditional template programming by performing a validation step in the modeling tool before calling the template engine. The tradeoff is however that by doing this validation step, the modeling-tool becomes related to the templates and the mapping rules it encodes, which may be undesirable.

Naming

It is very tempting to map model class names directly to Java class names and similar for attributes. In practice this leads to undesirable restrictions on the naming in the model. For example, it would not be allowed to have a class *Class* or something named *package* or any other reserved word in Java (if that is the target of your code generation). In addition, one model class may map to many generated classes or items. For example, in a J2EE EJB environment, you would maybe generate a *Home* and a *Bean* and an *Impl* class and maybe a database table for every model class. One option is to expand your model to include all these classes according to specific patterns. This will explode the size of your model however, and if the transformation is not reversible, you will be stuck with un-synchronized, un-maintainable models.

For FUUT-je we have chosen to use suffixes for classes and prefixes for attributes. For example, using the FUUT-je Swingset, for the model class *GuestEntry,* the classes *GuestEntryData, GuestEntryPanel, GuestEntryTable* and *GuestEntryDelegate* are generated. Attribute names are prefixed with *att.*

Another issue is the *naming of attributes which hold relationships*. In general the attribute should be named after the name of the relationship, it may default to the name of the class (with first letter put in lowercase) when there is only one relation between two classes. If a relation is a one to many relation, we may need to find out the *plural* of a name. This can default to *name+s* in English, but of course that is not always right. *Countrys* and *Adresss* looks ugly in code (although it does not influence the right execution of it).

Naming of *XML tags* deserves special mention. Tag names for XML are much more permissive than for Java. In practice I have encountered things such as <address-book>, to be mapped to a class *AddressBook* etc. If possible you need to make the modeling tool user friendly enough that this mapping will occur automatically. FUUT-je does this for you, unless you specify deliberately something different. It is not possible to solve this completely with template programming. The templates have to be aware of the mapping rules however.

Database Mapping

A special case of naming issues is the mapping class- and attribute names in a model to *table-* and *column names* in a relational database. The database may pose *length restrictions*, the database administrators may want to enforce *naming conventions*, you may have to map to an *existing database*, the mapping may not be *one-one*, you have to do special mappings for *inheritance structures*, .etc The algorithms needed may become quite involved. It may also be necessary to build your meta-model in such a way that it allows for specifying specific mappings of names.

Mapping the *types* of the attributes to the corresponding column types form the next challenge. For example, a Java attribute may have the type *String*. In a database there are many types of string: fixed length, variable lengths, long and very long. For each attribute you need to be able to specify how it will map, by giving it a specific stereotype, or by providing an attribute in the meta-model where you can enter the exact mapping. Another notorious problem is *dates*. The choice here are DATE, TIMESTAMP, or an INT corresponding with a Unix timestamp. Not all databases support the same types.

Cross-Cutting Concerns

In FUUT-je it is possible to specify a *group* name for an attribute. Attributes with the same group name are placed in the same *tab-panel* with the group name as its label. This is an example of a cross-cutting concern, where it may be needed to cycle through all model items several times before you can start generating code. Modeling this in another way than as an attribute for the model class *Attribute* would make the meta-model more complex. As a consequence the template that generates the tab-panels has become more complex. Tradeoffs are not always easy to make here.

Another cross-cutting concern is building a parameter list. For example you could provide a class constructor with a set of arguments that provide values for each (or a subset) of its attributes. Because you cannot place a ',' after the last argument, building these lists are a true pain. The best solution is to provide special methods in the utility class to do this.

Preservation of Manual Code

When modifications are made to the model, it may become necessary to regenerate the code. If you made changes to the code, the may be lost if you do not take special precaution. The first code generator I used (for the IBM San Francisco framework) forced you to cut and paste modifications back into generated code. Moreover, the code was such, that it would not even compile right after regeneration. This became very painful very quickly. Later code generators and generation wizards all use the same technique of allowing you modifiable code blocks that will not be touched by code generation. During template processing it is not possible or it would be very difficult to look into the previously generated file.

One solution is that the modeling/generation tool reads the code and disassembles it before the new generation is started. A disadvantage of this approach is that the generated files and the model need to be managed together and can get out-of-sync.

Another solution is (this is the approach taken by FUUT-je) to keep the modified code **in the model itself**. A special merge process is needed to load the modified code. It is much easier now to manage the code and structural changes, renames of classes, etc. are easier to perform.

A note on *reverse engineering* of code: Many claim that it is a drawback of generative programming that round-trip engineering is either impossible or prohibitively complex. We think however that this is an *advantage.* In the same way that you never look at the assembly code (unless you are a compiler writer) and try to make C or C++ or Java out of it, it is not useful to re-engineer Java code into a model. Which does not mean that the generated code should not be as readable as possible.

Related Work

Many efforts are undertaken currently to create tools that fit into the Model Driven Architecture (MDA)[18] paradigm or into the broader methodology of Model-Driven Software Development (MDSD)[7]. To take a comparative look at those, even considering only the open source ones, would take an article in itself.

So far, I did not see a comporative study to look at template languages intended for code generation, or a study that would consider the syntax and semantics of such a language scientifically.

Acknowledgement

Mark Kofman (kofman@kth.se) has pioneered the EMF version of the meta-model and has helped with the code generation extension.

Notes and References

[1] See www.eclipse.org/gmt for a description and status of the project.

[2] See http://www.eclipse.org/gmt/ and look for information on *FUUT-je*.

[3] See http://www.eclipse.org/emf

[4] http://www.w3.org/TR/xslt

[5] Ghica van Emde Boas, SF Business Component Prototyper, an Experiment in Architecture for Application Development Tools", the IBM Systems Journal, May 2000. The complete article can be found at: http://www.research.ibm.com/journal/sj/392/vanemdeboas.html.

[6] OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP, OMG document AD/2002-04-10. Available from www.omg.org.

[7] See www.mdsd.info for more information.

[8] In the white paper at http://www.softmetaware.com/mdsd-and-isad.pdf, this meta-model is referred to as a Domain Specific Language (DSL). We will not use it here. The term DSL is misleading because *domain* is usually associated with the *user* domain, i.e. **insurance** for insurance applications and **order management** for order management applications etc. We are talking here about *technical domains* such as **J2EE** or a **business component framework**.

[9] See http://c2.com/cgi/wiki?PlainOldJavaObject

[10] MOF is the meta-modeling language for UML. ECore is the meta-modeling language for EMF.

[11] See http://eclipse.org/articles/Article-JET/jet_tutorial1.html

[12] A summary of the discussion can be found at news.eclipse.org/eclipse.tools.emf, in an append by Frank Budinsky, on 7/20/2004. See also http://jakarta.apache.org/velocity/casestudy1.html. This link is a JSP to Velocity comparison.

[13] Velocity can be found at http://jakarta.apache.org/velocity/index.html.

[14] http://www.omondo.com

[15] http://www-306.ibm.com/software/awdtools/developer/java/

[16] http://www.eclipse.org/emf/

[17] *Private* attributes are considered **harmful** by the author of this article. We have been hit many times not being able to override a method properly because the super method used a private attribute.

[18] http://www.omg.org/mda