Best Practices for Model-to-Text Transformations

Version 1.0, August 18, 2006

Markus Völter, voelter – ingenieurbüro für softwaretechnologie, Heidenheim, Germany, www.voelter.de, voelter@acm.org

Bernd Kolb, kolbw@re, Heidenheim, Germany, www.kolbware.de, b.kolb@kolbware.de

1. Introduction

In Model-Driven Software Development, the generation of textual artifacts – often source code – plays an important role. However, often, code generation is seen as the "less important brother" of model-to-model transformations and is consequently treated as a second-class citizen. However, most developers come into MDSD through "simple" code generation and in most cases, the last step of a transformation chain is actually a code generator. It is therefore important that the generator is up to the challenge of generating non-trivial software systems.

In this paper, we will report on best practices for code generation. The authors are part of the openArchitectureWare team and have significant experience wrt. code generation, from the perspectives of the tool builder, as well as from the perspective of the tool user (i.e. generator developer).

We think that it is essential that a future Eclipse model-to-text transformation project considers these best practices.

2. Background - openArchitectureWare

openArchitectureWare (oAW) is a framework for model-driven software development. Currently, the tool is available in version 4.1, released on August 16, 2006. oAW comes with a host of features necessary for MDSD, including M2M transformations, declarative constraints checking, a workflow engine, adapters for the XMI of a variety of UML tools, EMF integration, nice Eclipse IDE integration (with custom editors and static error checking) as well as a proven template language for code generation called Xpand. Specifically the code generation language has been available for a number of years, so there is considerable industry experience available for that language. In this paper, we want to report on the experiences made with this part of oAW.

3. Best Practices

You will not that many of the best practices outlined below seem obvious from the perspective of a language designer. However, many code generation tools are not designed by language designers (we aren't language designers either ☺), but by tool builders who want to pragmatically get a code generation solution. For these people we find that the best practices are useful to read.

Code generation is not the simple replacement of wildcards with model-specific text. It is also not the same as a web templating system. Therefore, it is essential to provide a custom language (a DSL for code generation) and not to adapt some other technology.

Especially if you build non-trivial generators, you must provide means to structure templates into modules, reuse parts of templates and adapt templates in specific ways (e.g. if you want to generate code for various similar, but not identical platforms). The best practices below help.

3.1 Ignore Conrete Syntax

A code generation language must not access the concrete syntax of the model from which it generates code. Working on the concrete syntax makes matters more complicated than necessary and also binds the templates to the concrete syntax of the models. Transforming XMI into code using XSLT and the like is therefore out of the question. Consequently, models are typically represented as object graphs (as e.g. in EMF). However, a similar problem arises: accessing the model directly by calling methods from the template is more or less the same problem, since you cannot access model properties that are not represented as Java class structures. Therefore, always introduce an intermediate type system against which the templates are written. Different implementations of that type system can map the calls to Java classes, XMI files or other (maybe dynamic) representations.

3.2 Dynamic EMF

A specific consequence of the previous paragraph is that you should be able to work with dynamic EMF models! Regenerating the implementation classes for Ecore-based metamodels can be really annoying (for various reasons: the number of clicks required, and the sometimes rather strange behaviour of the code generator wrt. to merging). If you work with dynamic instances – and if you can generate code from them – these problems go away.

3.3 Powerful Expression Language

There are three kinds of model access when writing templates. The first and simplest one is the property access, e.g. if you want to access the *name* of an element in order to name a class you want to generate. Number two is model navigation, often crossing several references, to-one and to-many. The third kind of model access is selection of some kind of subset of a number of elements. In order to make these things painless, you need a simple and powerful expression language. In order to access a property, you should only have to write the properties name (in between some kind of escape symbol, see below), not something like *context.get("something").select("someProperty")*. In order to navigate templates easily, you expression language must be able to "collect" leaves of a tree, and e.g. apply properties to all elements of a collection, so you can write things like *class.operations.parameters.type.qualifiedname* in order to collect a set of all names of all types of all parameters of all operations of a class (e.g. in order to generate Java import statements). Finally, in order to select parts of a model, you'll need higher level functions such as map, select, collect, etc.

OCL, or a suitable subset of OCL can be used as the expression language provided it can be embedded natively into the templates, and not as a quoted (and non type-checked) string.

3.4 Modular Templates

As soon as you have a significant number of template LOC, you need ways to reuse parts of those templates. The canonical example is the method signature. It is typically required in many places in various templates. In order to not have to rewrite the respective template code over and over again, it is useful to factor out a piece of template code into a "subprogram", i.e. a template that you'll call from several places. It is not useful if you are required to have a 1:1 relationship between the file you want to generate and the template that generates it. You need a finer grained way of modularizing templates.

3.5 Nice Syntax

Yes, templates are programs that should be read by humans! As such, readability is an issue. Template languages always need some kind of escape character, to "toggle" between code that should be generated and code, that is used to control the template engine and access the model. Two things are important:

- The character(s) used for escaping needs to be "nice" and not an unreadable conglomerate of symbols such as <!&&
- It is important, at least these days, that you can generate XML without escaping all the < and > in the XML! So template engine escapes using < and > are problematic.

Addressing these two issues can be done in two ways: You can use characters as escapes that are not typically used in software and maybe aren't even on your keyboard. oAW uses the french quotation marks ("guillemots"). oAW provides special keyboard shortcuts for them in the Eclipse-based editors.

Second, for the situations where you need to generate you escape characters literally, you might want to provide a second set of escapes. A nice example for this approach (from another domain) is strings in HTML: You can either use '' or "" to delimit them. If you want to have a string that contains '' or "", you can use the other pair to delimit that string in the code.

3.6 OO Templates – the *this* reference

A specific – modularized – template is typically executed in the context of a specific model element. It is convenient to be able to reference that element by an implicit *this* reference. So, whenever you access a model property without any further qualification, it is by default resolved on the current *this* object. As a consequence, a template definition always includes the metaclass for which it is defined (just as a method in an OO program that is always attached to a class).

So, if you are in the context of an *Entity* element (something defined in your metamodel), you might have a template that generates a implementation for entities. The signature for the template might be defined as

```
<<TEMPLATE javaClass FOR mymetamodel.Entity>>
```

The expression <<*name*>> in the template body resolves to the name of the particular *Entity*. If you have a separate template that can generate method signatures, then this other template might be defined as

```
<<TEMPLATE javaSignature FOR mymetamodel.Method>>
```

You can call that template from the original one, e.g. by writing

```
<<EXECUTE javaSignature FOREACH methods>>
```

where *methods* is the Entity's property that returns the set of all methods for the particular Entity. Each of the elements in the foreach statement becomes the *this* object of the called template.

3.7 Template Polymorphism

Once you have "object oriented templates" in place (as described in the previous section) you can now also support template polymorpism. This relieves from writing all kinds of type-if's (using instanceof, oclIsKindOf and the like) in the templates. Type-if's are bad because, just as in normal OO programming, you have to revisit all the type-if's if you add a new subclass. Using template polymorphism, you can write things such as

```
...
<<EXECUTE methods FOREACH properties>>
...

<<TEMPLATE methods FOR ReadOnlyProperty>>
  ...generate only a getter method...
<<END>>

<<TEMPLATE methods FOR Property>>
  ... generate getters and setters ...
<<END>>
```

This example assumes that *ReadOnlyProperty* is a sub-metatype of *Property*. If at some point you'll introduce a *DerivedProperty* metaclass, you only need to provide specialized templates (*TEMPLATE ... FOR DerivedProperty*) where the code for derived properties differs from "normal" properties.

Again, this best practice helps to keep your generator code small and maintainable, especially in the face of complex metamodels or complex target code generation requirements.

3.8 Static Type Checking & IDE support

An almost religious discussion among language designers is the question about typing. There are various alternatives: strong/static/weak/dynamic/duck etc. The real important aspect is that, when writing templates, you can use static type checking in the editor. oAW 3.x used dynamic type checking, i.e. you could basically call any property on a model element, and only at runtime you'd know whether the property actually existed (and was of the type the rest of your expression expected). In oAW 4.x the editor supports static type checking, i.e. when writing the templates, you will receive error messages in real time if the metamodel underlying the template doesn't support a certain property. Also, because we have all the type information available in the editor, we can provide metamodel-aware code completion (see Figure 1).

Why are these two aspects important, especially in code generation (and model transformation) languages?

- The type system you work with changes from project to project (since the metamodel is typically different in all the projects). Providing developers support in working with the metamodels, especially if it's a complex and non-intuitive one such as the UML2 metamodel, can boost productivity.

- A generator run can take quite a while. If all the typing errors are only found once the generator runs, finding rrors and fixing them can be a tedious task.

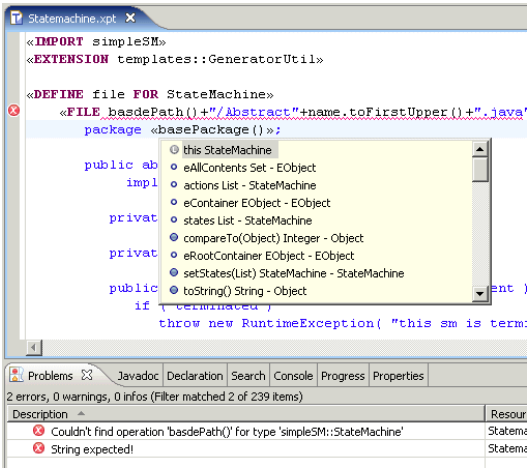

Figure 1: oAW's template editor

3.9 Extensions

Whenever you generate code for a particular platform, you'll need additional properties in your metamodel. These are typically derived from other properties, specific for the particular target platform and often too intricate to be expressed as templates. However, adding these additional properties to your domain metamodel is not a good idea, because the metamodel should represent the problem domain – and therefore shouldn't be polluted with additional, solution-space specific properties. This becomes especially ovious if you generate code for several target platforms from the same model. You'd have all the additional properties for all the platforms in the metamodel. Therefore, you should provide a means of adding additional properties to existing metaclasses using *external* definitions. This is somewhat like open classes: you can add features to a meta model element without modifying its original definition. If you have a nice expressioin language in place, you can use that expression language to define the body for these additional properties.

Note that this best practices is not just about things being "nice" or "not nice". Assume a third party wants to build generation templates for an additional target platform; withouth extensions, they'd have to modify the original metamodel which might not be feasible for technical, legal or testing-related reasons.

Some people argue that this best practice would not be necessary if you'd simply do a M2M transformation into some platform-specific representation, and then generate code from that. The M2M transformation could add these additional properties to the PSM, and the PSM metamodel would contain them. While this is true in principle, it is often not pragmatic to add a separate M2M step in before code generation, just because you need a couple of additional properties.

3.10 Template AOP

This one is an advanced feature, but it has "saved our lives" a couple of times. Assume you're generating code for embedded systems. The code you're generating must be able to run on several different hardware architectures. The generated code is C. So, the code for the various platforms is mostly the same, but there are small (and not so small) differences scattered through the code. The classical solution is to have templates that look somewhat like the following:

```
...
all kinds of C stuff here that is common to all
the supported platforms
...
<<IF platform==”68HC11”>>
  ...68HC11-specific code here
<<ELSEIF platform==”8051”>>
  ...8051-specific code here
<<END>>
```

As you can see, the platform specific aspects are scattered through the code – they cross-cut the template structure. So, as we all know, AOP can help to localize and modularize cross-cutting concerns. Therefore, what we need is AO support in template languages. This allows us to put all the "general" stuff into the core templates, and then weave the platform-specific stuff in – all the platform-specific template code will be collected into aspect templates.

In openArchitectureWare we support a special template definition syntax:

```
<<AROUND pointcut FOR type>>
  to-be-generated-code
  <<targetDef.proceed()>>
<<END>>
```

Note that the pointcut supports several wildcards and the type respects the polymorphism explained above.

4. Conclusions

The best practices are what we consider essential for practically usable and scalable templates languages. Some of them might be obvious, others not so. Also, some of the best practices are useful for MDSD in general, and not limited to code generation (e.g. they might be useful for model-to-model, too).

If you take one point from this paper, it should be: template languages aren't second-class, trivial text processorts. They should be treated as actual languages – yes, special purpose languages (or DSLs ☺), but they're languages. We can apply many of the lessons learned from programming language design to template languages, and we should do so!

5. Acknowledgements

We would like to thank the other members of the openArchitectureWare team for the contributing to the tool; specifically, we want to thank Sven Efftinge for reviewing the paper and contributing ideas.