Principle Structure and Architecture of a Code Generator

Andreas Schmidt[*][†]
[*] Faculty of Computer Science and Business Information Systems,
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
Email: andreas.schmidt@hs-karlsruhe.de
[†] Institute for Automation and Applied Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: andreas.schmidt@kit.edu

***Abstract*—Code generators often have something mystical about them. Especially undergraduate students, who can still remember their first steps in programming, become in awe when they hear the term "Software Generator". The paper is an attempt to take this awe away from the students and to show them, by means of a very simple example implementation with well-known tools and technologies, that software generators are not witches' work, but a powerful, but easily understandable tool to support the software development process.**

Keywords–Code generation; template system; (meta) model; model transformation.

I. Introduction

In this paper, the general structure of a code generator is presented. It is intended as additional material to the tutorial with the title "Code generation for Database Developers" which is also given by the author at the DBKDA-2020 conference in Lisbon [1]. The principle structure and architecture of a general purpose code generator will be explained with the help of a simple example implementation, using well known tools and techniques. The procedure is from the backend of the generator, over the kernel to the frontend. The advantage of this approach is that one can see the final result (the generated code) right at the beginning and then deal with the details to achieve this result. The Template Engine of the generator, the internal metamodel, the import module, the external metamodel and the transformation of XMI (**X**ML **M**etadata **I**nterchange) - the standard exchange format for models - into the previously developed metamodel are then presented.

A. Principle Function of a Generator

The principle mode of operation of a generator is shown in Figure 1. The generator obtains as input an abstract model description and a set of transformation rules, which describe the transformation of the abstract model into the source code. It is crucial that the model is formal and the model description is available in a form that abstracts from implementation specific details. Through one or more model transformations, the implementation details are added to the target platform. This achieves a separation between the business logic and the technical aspects of the target platform.

B. Advantages of Generative Software Development

Herrington [2] names four main advantages of generative software development, which are to be presented in the following briefly.

1) Quality: The quality of the software is determined by the transformation rules. Over time, these rules gain more and more quality, so that the quality of the generated source code increases. The automatic transformations avoid careless mistakes. If individual transformation rules are faulty, these errors occur at all places that use the faulty transformation rules and are therefore easy to find and correct. Furthermore, when developing the transformation rules, more thought is given to the architecture of the application in advance than when starting directly with the coding. The previously considered architecture is then consistently implemented in the complete source code by the transformation rules.

2) Consistency: Source code generated by transformation rules is very consistent regarding naming, calling conventions and parameter passing, so that it is quite easy to understand and use. This offers a starting point for further possible automations. Cross-sectional functionalities such as logging or error handling can be defined centrally and thus be adapted to changing requirements at any time (analogous to aspect-oriented programming).

3) Productivity: Productivity in application development increases. Even if only so-called infrastructure code is generated, which is often considered to be the boring part of programming, more time remains to take care of the actual (exciting) application logic. Furthermore, it is possible to react faster to design changes or change requirements, because only the corresponding transformation rules have to be adapted and the application has to be regenerated.

4) Abstraction: The model represents an abstract description of the application to be realized. The strict separation of domain-oriented logic (model) and technical aspects (transformation rules) reduces complexity. This, in turn, allows for a

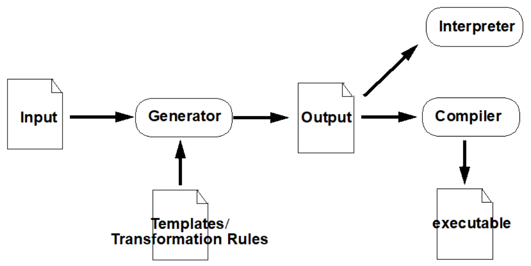


Figure 1. General Architecture

better integration of domain experts within the development project, as they can be involved in the development of the model. Another advantage is the easier transition to a new technology, since only the transformation rules have to be adapted, since the domain-oriented logic of the model remains valid. On the other hand, transformation rules once developed can be reused in other applications.

The remaining paper is structured as follows. In Section 2, we will discuss which artifacts can typically be generated. In Section 3, the development of the generator is presented in detail, divided into backend, kernel and frontend functionality. In Section 4, the automation of the single steps using the Unix tool *make* is discussed. Section 5 concludes with a discussion of possible extensions for the generator prototype.

II. What can be generated?

The goal is the partial or complete generation of the source code for an application to be realized. The degree of automation usually ranges from 20% to 80% of a application. Higher levels of automation are possible but often not useful, because this would make the generator much more complex than implementing the missing 20% of the software by hand [3]. For web-based applications, a degree of automation of about 60-70% can often be achieved. Typical parts of an application that can be generated include the following areas:

- Database schemas
- Access layers for databases
- User interfaces
- Parts of the application logic
- Documentation
- Configurations (e.g., in combination with frameworks like Struts, Spring, Hibernate, etc.)
- Tests (unit tests, constraint tests, generation of mock objects, load tests, etc.)
- wrapper
- Import/Export Modules
- etc.

III. Development of the Generator

In the following, a multipurpose generator is to be built up by the simplest means. This is done exemplarily with the programming language PHP [4]. The reasons for using PHP are the following: PHP is a macro language and can therefore also be used as a template system, which can be used for the definition of the mapping rules. In addition, there are also special template languages for PHP, which can be used for this purpose. Due to its primary field of application as a language for creating dynamic websites, PHP is characterized by its powerful string handling. This is also useful for generating source code. Furthermore, there are many free libraries available for PHP (PHP Extension & Application Repository - PEAR). Other languages suitable for this task are Perl, Python and Ruby.

Besides PHP the following tools/technologies are used:

- An XSLT [5] or XQuery [6] processor to transform XMI into a simpler meta-format
- The Unix tool make [7] for automation of the entire workflow

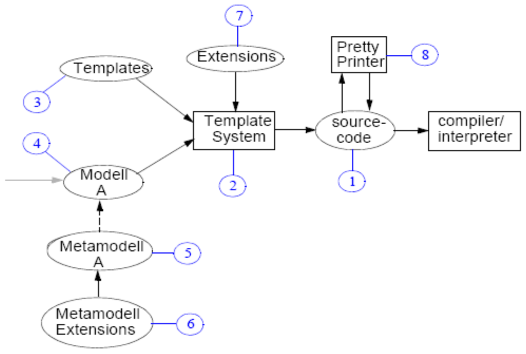


Figure 2. Generator Backend

- The Smarty template engine [8].
- An UML modeling tool for graphical modeling (i.e., [9]).

A. *Scope of Functions and Expansion Options*

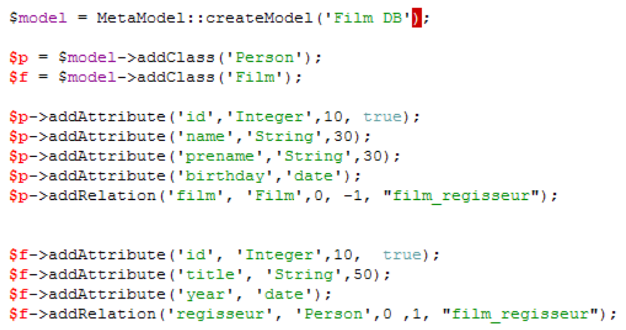
The functionality of the generator to be developed is limited to the generation of artifacts based on the information of a simple class model. This does not represent a limitation for the basic architecture. In Section V it is shown how the generator can be extended to process further model elements (e.g. state transition diagrams).

B. *Generator Backend*

The basic design of the generator jaw is shown in Figure 2. The backend is responsible for the actual generation of the source code (1). For this purpose, a template system (2) is used, whose task it is to create clear mapping rules from the model to the target language by separating the dynamic and static parts. For this purpose, the template system uses as input on the one hand the so-called templates (3), in which the transformation rules for the generation of the source code in the form of static text and simple control flow elements, such as loops and conditional statements as well as placeholders for the information originating from the model are stored, and on the other hand the model (4) on which the application to be generated is based, which contains the dynamic parts of the source code to be generated.

In the concrete case, the model is available in the form of an arbitrarily complex object network, which describes the artifacts to be modeled such as classes, attributes with types, as well as relationships. The model is also based on the so-called meta model (5), which defines the modeling possibilities in the form of classes and the associated methods. This metamodel is realized by means of PHP classes. Figure 3 shows a code snippet for defining a model using the Metamodel API. In the code section, the two classes ”person” and ”film” are defined with their attributes and furthermore the relationship ”film_director”, which models a 1:n relationship between film and person.

An example of a Smarty template is shown in Figure 4. The example shows a template for generating the database schema. Language elements of the template language are indicated by `[@ ... @]` brackets. Available language elements include loops, conditional statements, variable assignments, calling



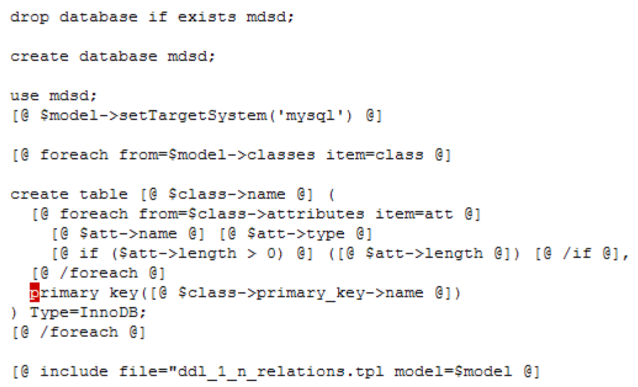
```
$model = MetaModel::createModel('Film DB');

$p = $model->addClass('Person');
$f = $model->addClass('Film');

$p->addAttribute('id','Integer',10, true);
$p->addAttribute('name','String',30);
$p->addAttribute('prename','String',30);
$p->addAttribute('birthday','date');
$p->addRelation('film', 'Film',0, -1, "film_regisseur");


$f->addAttribute('id', 'Integer',10,  true);
$f->addAttribute('title', 'String',50);
$f->addAttribute('year', 'date');
$f->addRelation('regisseur', 'Person',0 ,1, "film_regisseur");
```

Figure 3. Programmatic Model Definition



```
drop database if exists mdsd;

create database mdsd;

use mdsd;
[@ $model->setTargetSystem('mysql') @]

[@ foreach from=$model->classes item=class @]

create table [@ $class->name @] (
  [@ foreach from=$class->attributes item=att @]
    [@ $att->name @] [@ $att->type @]
    [@ if ($att->length > 0) @] ([@ $att->length @]) [@ /if @],
  [@ /foreach @]
  primary key([@ $class->primary_key->name @])
) Type=InnoDB;
[@ /foreach @]

[@ include file="ddl_1_n_relations.tpl model=$model @]
```

Figure 4. Template for generating the Database Schema

other templates and calling properties and methods of the meta model.

In order to make the creation of the templates as simple and clear as possible, it is often helpful to extend the meta model (6) or the generator (7) for certain specific language constructs of the target language instead of formulating these language constructs within the templates (3).

C. Generator Kernel

The actual heart of the generator is represented by the generator kernel. Figure 5 shows the transformation and validation components of the generator. The methods of the metamodel already monitor a number of constraints in the model, for example that the classes have different names and that the attributes must be of certain predefined types. However, there are also constraints that cannot be enforced in this way, for example the constraint that each class must have a primary key or that certain attributes/relationships must exist for each class. For this purpose, the generator provides an interface for the formulation of validation rules (11). These are implemented in the form of PHP methods (12). An example of such a method is shown in Figure 6. This method monitors that each class must have a primary key. The methods also work like the templates on the properties and methods of the Metamodel API.

Furthermore, model transformations (13, 14) can be formulated. In the simple case these are transformations within the same metamodel (13). For example, additional administrative information (created_at, created_from, etc.) can be added to a model for each class (see Figure 6. For the formulation of

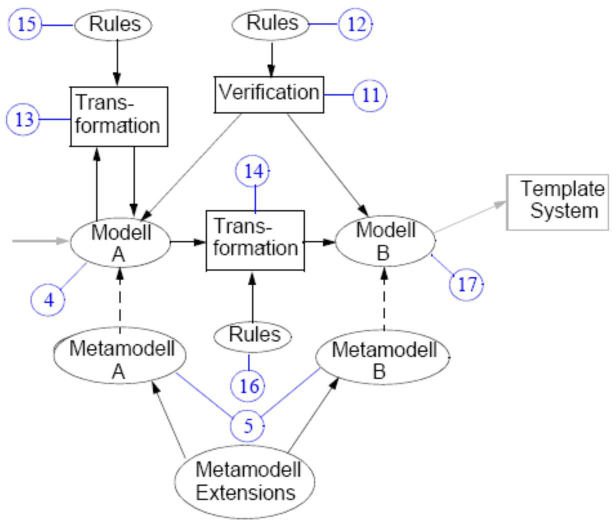
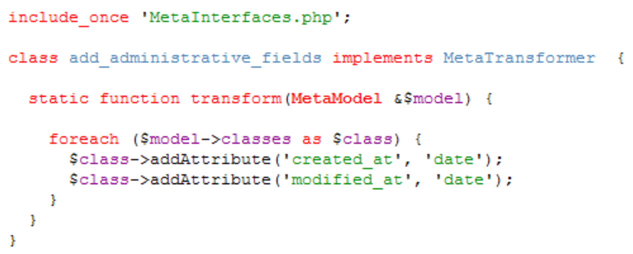


Figure 5. Generator kernel



```
include_once 'MetaInterfaces.php';

class add_administrative_fields implements MetaTransformer  {

  static function transform(MetaModel &$model) {

    foreach ($model->classes as $class) {
      $class->addAttribute('created_at', 'date');
      $class->addAttribute('modified_at', 'date');
    }
  }
}
```

Figure 6. Example Model Transformation

transformation rules, the generator also provides an interface, which allows the formulation of transformations in the form of methods (15). Furthermore, transformations (14) to another metamodel (16) are also possible (e.g., to a metamodel with the concepts table, attribute, foreign key, constraints, etc.).

D. Generator Frontend

1) Model Import: Up to now, models can only be built using the methods available in the metamodel, i.e., programmatically through a series of API calls. However, this is not desirable and so an XML format (Figure 7, point 21) is defined in an extension of the generator, which allows the formulation of the model as an XML file (22). In this case, the meta model is represented by the DTD (21) and thus defines what can be formulated in the model file. In an import process (24) the DOM tree of the XML file is then created and a transformation (24) to the internal model (4) is carried out by the methods available in PHP for processing XML, i.e., the corresponding methods of the internal meta model are called during navigation through the DOM tree and thus the internal model representation (4) is built up. Optionally, the XML file (22) can be modified by means of an XSLT transformation (25) before import. Meaningful transformations on this level are, for example, the addition of further attributes or primary keys, if these have not already been specified in the UML model.

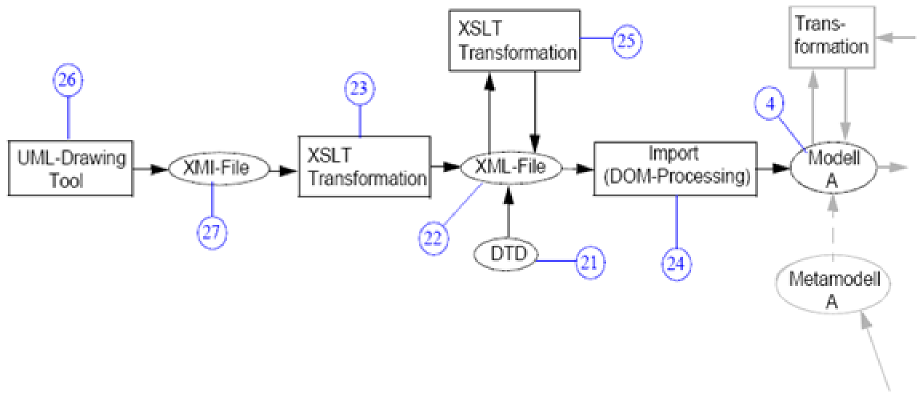


Figure 7. Generator Frontend

2) Connecting the Frontend: The connection of a UML modeling tool (26) is realized by the XMI export interface provided by most tools. XMI is an XML-based, standardized exchange format for UML models. To connect to the generator, all that is required is the development of an XSLT stylesheet (27), which extracts the relevant information from the XMI file and transforms it into the previously developed XML format (22). It is also possible to do without the own XML format and import the XMI file directly from the generator. The disadvantage of this variant, however, is that XMI is an extremely "chatty" format and the import and transformation into the internal meta model is much more complicated than via the detour of the intermediate format.

IV. Automation

From the creation of the UML model to the export as XMI file, the XSLT transformation into the generator's own XML format, the model validation/transformation, the actual code generation based on the created templates, and any subsequent source code formatting (Figure 2, point 8), a complete generator run represents a complex workflow consisting of many individual steps and dependencies. To automate this, the development tool "make" is used here. It allows the formulation of sequences of work steps as well as dependencies, which then cause a conditional execution of parts of the workflow.

V. Extension of the Generator

The generator introduced so far supports the generation of artifacts, which can be derived from a simple class model. In the context of the lectures carried out at the University of Applied Sciences Karlsruhe - Technology and Economics as a compulsory elective subject in the field of Business Informatics as well as further tutorials [10], [11] it was shown that a very high learning effect can be achieved by letting the participants extend the generator by additional diagram types. In contrast to the initial introduction of the generator, a forward-looking approach is suitable for the extension, i.e., starting from an XMI file generated by a modeling tool, the own XML format is extended and the corresponding XSLT transformation is adapted. Subsequently, the internal meta model must also be extended by the corresponding concepts and the import filter must be adapted accordingly. The last step is to create additional templates or to extend the existing templates. As extension for example the addition of state transition diagrams or the addition of the inheritance concept for class diagrams is suitable. A further instructive extension is the mapping of the present meta model to another meta model, which represents the concepts of relational databases and the subsequent adaptation of the templates.

VI. Conclusion

The presented framework shows in a simple way how a software generator works. Due to its easy extensibility, it can be adapted to own needs very easily. However, it is not intended to compete with existing tools but is mainly used in teaching. Nevertheless, it can be used to create own generators for applications where it is not worthwhile to learn a commercial or freely available tool.

References

[1] A. Schmidt, "Code Generation for Database Developers. Twelfth International Conference on Advances in Databases, Knowledge, and Data Applications - DBKDA ," Lisbon, Portugal, 2020, URL: https://www.iaria.org/conferences2020/ProgramDBKDA20.html, last accessed: September 2020.

[2] J. Herrington, Code Generation in Action. USA: Manning Publications Co., 2003.

[3] T. Stahl, M. Voelter, and K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006.

[4] K. Tatroe and P. Macintyre, Programming PHP, O'Reilly. Sebastopol: O'Reilly, 2006.

[5] T. Doug, XSLT. Sebastopol: O'Reilly, 2008.

[6] P. Walmsley, XQuery: Search Across a Variety of XML Data. Sebastopol: O'Reilly, 2007.

[7] R. Mecklenburg, Managing Projects with GNU Make. Sebastopol: O'Reilly, 2004.

[8] L. Gheorghe, H. Hayder, and J. P. Maia, Smarty PHP Template Programming and Applications. Packt Publishing, 2006.

[9] "argo UML," URL: https://github.com/argouml-tigris-org/argouml, last accessed: September 2020.

[10] A. Schmidt, "Supporting the development of web-based applications with lightweight software generators. Third International Conference on Internet technologies and Applications - ITA09," Wrexham, Wales, 2009, Tutorial session.

[11] A. Schmidt, "The power of regular expressions in the software development process. International Conferences on Informatics 2010 Software Engineering and Applications SEA 2010," Marina del Rey, USA, 2007, Tutorial session.