# An Approach to Conveying Fundamentals in the Field of Model-driven Software Development

Andreas Schmidt[1,2], Markus Dickerhof[2] and Daniel Kimmig[2]

[1] University of Applied Sciences, Karlsruhe
Faculty of Informatics and Business Information Systems
76133 Karlsruhe, Moltkestraße 30, Germany

[2] Karlsruhe Institut of Technology, Institute for Applied Computer Sciences
76344 Eggenstein Leopoldshafen, Hermann v. Helmholtz-Platz 1, Germany

**Abstract.** A didactial approch for teaching model-driven development is proposed in this paper. The main idea is to focus on conveying underlying concepts rather than managing a concrete tool. This objective shall be reached by the development of a simple generator. For this reason the whole process from graphical modeling to the code generation proper is traversed twice, the first time from back to front to introduce the main concepts of a generator engine and in a second pass starting from the beginning to extend the generator with additional functionality. The lecture will then be completed by transferring the knowledge learnt to a concrete generator tool within the framework of a simple exercise and by a presentation.



## 1 Introduction

Model-driven software development has been existing for many years already, but recently interest increased strongly due to the MDA initiative of the OMG. Consequently, the industry's demand for graduates qualified in this field has grown. We therefore considered it reasonable to include model-driven software development in the education of our students. The original idea was to take a concrete tool in order to illustrate the concepts of model-driven software development. However, this was associated with the risk of the students learning the use of a concrete tool, but neglecting the concepts. Transfer to other tools and the capability of developing own simple code generators would be missed out. For this reason, we decided to choose an entirely different approach. Attention will focus on conveying underlying concepts rather than managing a concrete tool. This objective shall be reached by the development of a simple generator based on tools that are mostly known to the students. At first, the capability of the generator will be limited to the generation of concepts that can be modeled in a simple class diagram, for example, classes in an object-oriented programming language, database schemes, the database access layer, and simple user interfaces. The complete workflow starting from graphical modeling by an external tool to transformation into an own meta model, to potential transformations within this meta model or optional transformation into another

Schmidt A., Dickerhof M. and Kimmig D.
An Approach to Conveying Fundamentals in the Field of Model-driven Software Development.
DOI: 10.5220/0003009700560067
In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development (ICEIS 2010)*, page
ISBN: 978-989-8425-10-2

meta model, to actual code generation is executed two times. In the first run, it is started at the end, i.e. with code generation, and worked successively to the beginning. In this way, the students see the result at the beginning (the source code generated) and can easily derive the necessity and capability of the upstream components. This derivation of requirements for upstream components is achieved by extending the initial task. At the end of the first run, a complete execution chain exists from modeling with an UML modeling tool, to several transformation steps, to the generation and formatting of the source code to be generated. The second phase is aimed at extending the generator with additional features, such as other UML diagram types, e.g. state transition diagrams. Contrary to the first run, it is proceeded in the other direction, i.e. the extensions are made starting from the output of the UML modeling tool, to the meta model, to code generation by templates.

Due to this double treatment of basic components of a code generator, in-depth understanding of the functioning of a generator is obtained, which by far exceeds the understanding obtained from learning a concrete tool. While the first run (from the back to the front) focuses on understanding the individual components, the second run (extension) concentrates on their interaction.

The lecture will then be completed by transferring the knowledge learnt to a concrete generator tool within the framework of a simple exercise and by a presentation. Particular attention shall be paid to the concrete implementation of the individual components.

**Contribution of the Paper.** The paper presents a successful approach to conveying fundamental knowledge in the field of model-driven software development. Doing this, it will be focused on the understanding of fundamentals rather than on learning a concrete tool. Concrete approaches, such as those applied within the framework of the MDA initiative, will not be conveyed. However, they may be referred to partly in the final presentations of the students.

**Structure of the Paper.** First the tools used shall be presented and reasons for their selection shall be given. Then, the workflow covered by the lecture shall be described in detail, based on simple, but continuously growing requirements on the system. Each learning step, will be clearly motivated by an concrete example or extension. At the end the paper will be concluded by a summary.

## 2 Used Tools

Due to the relatively short time (two contact hour) for the implementation of an own code generator, it was assured that the technologies/tools used are largely known to the students or that their use is not associated with an excessive learning expenditure. For instance, it is done without the use of a descriptive language for formulating constraints. Instead, the constraints are formulated in a procedural manner. At these points, it will also be referred to the fact that other, often more comfortable, but also more complex solutions exist. Understanding of the generator, however, is not limited by these simplifications. They enable the student to concentrate on the most important points.

**Programming Language.** The generator kernel should be implemented using a scripting language due to its generally high productivity and the good string handling. Possible candidates are Perl [1], Python [2], Ruby [3], and PHP [4]. We selected the language PHP. The main reasons lie in the fact that the students have already gained first experience from the use of this language during their studies and that PHP is a so-called macro language that supplies a template mechanism.

**Meta Model.** To accelerate learning, a small PHP library is supplied to the students, which represents the initial meta model and allows for the formulation of their models by an API (see listing 1.1). The meta model allows for the formulation of classes, attributes, and relations. In addition, the meta model supplies a number of methods or properties for the iteration over the classes and their attributes. The meta model represents a central component in the system to be implemented and is used or extended by the students at many points.

**Template Engine.** The template mechanism incorporated in PHP is used at the beginning of the lecture, but later on replaced by the explicitly available template engine ”Smarty” [5] that is widely used in the PHP community and considered to be highly mature. In particular, Smarty has a very good documentation [7].

**UML-modelling Tool.** The only requirement made on the modeling tool is that an XMI export format can be written. This is given by most modeling tools.

**XML.** Both the external model to be developed and the XMI model generated by the modeling tool are XML-based. For this reason, DOM, XSLT, and XPath tools are used. The PHP DOM-XML library is applied. Xalan [6] is used as external XSLT transformation tool.

**Workflow.** It is a long way from graphical modeling to various model transformations to the actual code generation step. Often, certain parts of this generation (mostly transformations) are accomplished within the framework of development. Consequently, they shall not add up to a monolithic block, but exist as external components with clearly defined interfaces, which are then assembled by make [8].

## 3 Didactic Approach

The following sections will highlight the didactical implementation of the concept presented above. As mentioned in the introduction, the generator will be set up in two phases. In the first phase, a simple generator with minimum functionality, but a maximum of relevant concepts will be implemented, starting from the back-end (the meta model and the template engine). In a second run, this generator shall be extended, starting from the front-end (UML modeling tool, XMI import, transformation to internal model). In a last phase an existing tool should be used to implement a concrete task, based on the knowledge gained in the first two phases.

### 3.1 First Run - From Back-end to Front-end

**Back-end.** The starting point is the actual code generation. For this purpose, a simple Java source text consisting of two classes and a simple main program is presented to the students. It is now the task of the students to analyze the source code for parts that represent recurrent concepts for all classes and, hence, can be generalized and parts that are class-specific. It is the objective to have a class-specific part that is as small as possible.

*Task of the Students.* Then, the students are asked to generate a minimum PHP program for the given java source text, which generates the code. Here, it is important to clearly separate both parts (class-specific vs. generally valid class concepts in Java). Listing 1.1 shows the typical result to be produced by the students.

**Listing 1.1.** Simple hard-coded generator for the generation of two java classes.

```
<?
$classInfo = array( 'Person'=>array('name'=>'String',
                                    'yearOfBirth'=>'Date'),
                    'Film'=>array('title'=>'String',
                                  'year'=>'int')
                    );
?>
import java.util.Date;

<? foreach ($classInfo as $name=>$attributes) { ?>
  class <? echo $name ?> {
    <? foreach ($attributes as $attName=> $attType) { ?>
      <? echo $attType ?> <? echo $attName ?>;
    <? } ?>
    public <? echo $name ?>() {}  //empty constructor
  }
<? } ?>

class Test {
   // main program follows here
}
```

*Lessons Learned.*

- Identification of class-specific concepts (name of the class, attributes, types) and of the generally valid concepts of a class (general language syntax).
- Mapping of both parts on the generator model and template.
- Use of the PHP macro language property to generate simple, clear templates (compared to print statements).
- Interaction between model and template.
- A simple meta model to define models may consist of language concepts, such as arrays and dictionaries.
- The formatting of the generated source code is not realised inside the templates, but by an external code beautifier.

*In-depth exercise:* Extension of the generated source code by additional getter/setter methods without extending the model part.

**Back-end/Generator Kernel.** In the next lesson, the model shall be extended to cover also relationships among classes. Our example is a $1:n$ relation between a film and its director or a $n:m$ relation between films and their actors.

While the $1:n$ relation between a film and its director can be integrated relatively easily in the syntax of our model (Listing 1.2, last line), considerable efforts are required for $n:m$ relations. When interpreting the data structure in the templates at the latest will the students reach the limits of feasibility.

**Listing 1.2.** Possible extension of the model to represent $1:n$ relations.

```
$classInfo = array('Person'=>array('name'=>'String'),
                                   'year'=>'int',
                                   'director'=>'Person');
             );
```

It is therefore reasonable to introduce a more powerful meta model to model the relations. Such a meta model may possess the API shown in listing 1.3 for the formulation of the model:

**Listing 1.3.** API of a more comfortable meta-model to describe the model from listing 1.1.

```
$model = new Model('Demo');

$film = $model->createClass('Film');
$file->addAttribute('title', 'String');
$file->addAttribute('year','int');

$person = $model->createClass('Person');
$person->addAttribute('name', 'String');
$person->addAttribute('dateOfBirth','Date');
```

The meta model shown above is made available to the students. The reason for this is that the meta model plays a central role in the further setup of the generator and, hence, should be identical for all students at the beginning. The original model will then be extended by the students.

*Task of the Students.* The task of the students consists in extending the meta model supplied by additional methods. The methods to be implemented shall be used within the templates and contribute to simplifying the generation of templates. In particular, generation of templates for the implementation of the new relation concept shall be simplified (e.g. when mapping the relations on instance variables and/or methods).

*Task of the Students.* In the next step, a generator shall be developed, which generates schemes for relational databases (e.g. for MySQL). While handling this task, students encounter the following problems/deficiencies:

1. In both generators, the model exists in hard-coded form. When changing the model, these changes have to be made in all generators.
2. The data types for Java and MySQL have different designations (i.e. string vs. text).
3. The generation of foreign keys within the templates is not trivial, which often leads to templates that are very difficult to understand.

Within the framework of the lecture, a number of possible solutions for the above problems will be presented to the students:

*Solution for Problem 1.* The templates are no longer hard-coded in the generator, but explicitly stored in own files. When calling up the generator, the name of the desired template file is given as a parameter. In PHP, this can be done simply as follows (Listing 1.4):

**Listing 1.4.** Template as parameter to the generator.

```
$template = $ARGV[1];
...
include "$template.tpl";
```

The generator can then be called up as follows:

```
php.exe simpleGenerator.exe java > gen/java/src/Test.java
php.exe simpleGenerator.exe ddl > gen/sql/DDL.sql
```

In analogy, the model or both the model and the template can be parameterized[3].

*Solution for Problem 2:* Several solution approaches can be conveyed to the students. In general, the students develop a mapping function on their own, which is then called up in the templates. For instance[4]:

```
<?= $name ?> <?= map($type, JAVA) ?>
```



Another possibility consists in extending the meta model by the respective methods (listings 1.6 and 1.5).

**Listing 1.5.** Explicit target language setting at each datatype occurence.

```
<?= $att->getType(JAVA) ?> <?= $att->getName() ?>;
```

**Listing 1.6.** Global target language setting at the beginning of each template.

```
$model->setTargetLanguage(JAVA);
...
<?= $att->getType() ?> <?= $att->getName(); ?>;
```

A third possibility is the use of a template engine. Generally, template engines possess mechanisms to fulfill such requirements. In the Smarty template engine used during the lecture, the filter concept may be used for this purpose (see listing 1.7). In this case a filter with name `java` has to be written, which than maps the generic datatype to the concrete java datatype.

[3] which does not necessarily make sense when considering the minimum functionality of our generator which consists of two include statements only in our case

[4] In PHP, `<?= $variable ?>` is a short form for `<? echo $variable ?>`

**Listing 1.7.** Use of a filter within an explicit template system.

```
<? $att.type|java ?> <? $att.name|java ?>;
```

At this point of the lecture, basic concepts of a template engine and its concrete implementation are presented using the Smarty template engine as an example. To familiarize with the template engine, the template mechanism integrated in PHP for generator development shall now be replaced by the Smarty template engine. Apart from the practical testing of e.g. the filter mechanism, transition to this engine also results in a redesign of the templates generated so far and often of the complete application architecture, which promises to considerably improve the code. This applies in particular to the last problem expressed:

*Solution of Problem 3.* The approaches from the students to implementing foreign keys and relation tables rely often strong on a lot of PHP code inside the templates rather than by extensions of the meta model by suitable methods, which does not really enhance readability and maintainability. By using the external template engine, the students are forced to include methods (e.g. `$class->get1NRelations()`, `$relation->->getForeignTable()`)) in the meta model due to the limited vocabulary or to make a modularization using the respective Smarty concepts, which leads to much simpler templates in both cases.

*Lessons Learned.*

- The generator is given a model and a template as input.
- A more powerful meta model facilitates the formulation of the model and the generation of the templates.
- The extensions in the meta model generally are dependent on the target language.
- Template engines possess a very reduced vocabulary, but various possibilities of extensions or adaptations to the problem field.
- The writing of simple and clear expressed templates is essential for their extensibility and maintainability.

**Generator Kernel.** As the generator functionality so far has been limited mainly to the setup of the model by the meta model and the invocation of the template engine, the generator shall now be extended by major functionalities, such as model transformation and model verification.

The motivation is a concrete task again: All classes shall be extended by administrative attributes. These are the fields:

```
createdAt: Date
lastModifiedAt: Date
```

Of course, this task can be solved easily by extending the model by the respective fields, but this work is very monotonous and error-prone and can be automated easily.

*Task of the Students.* It is the task of the students to write a function that executes this process automatically. At first, this does not sound easy to the students, but they will be pleased when they will have found the solution (listing 1.8):

**Listing 1.8.** Intra model transformation.

```
function addAdministrativeFields() {
  foreach ($this->getClasses() as $class) {
    $class->addAttribute('createdAt', 'Date');
    $class->addAttribute('modifiedAt', 'Date');
  }
}
```

*In-depth Exercise.* Adding of relations to user entities in order to indicate who created/last modified the data set.

While the above example is a transformation within the same meta model, the generator to be developed shall also allow for transformations from one to another meta model. For this purpose, another simple meta model is made available to the students. This meta model contains the concepts *schema*, *table*, *primary key*, *attributes*, *foreign key*, ... (Listing 1.9 gives an example of the usage of this model).

*Task of the Students.* It is the task of the students to accomplish a model transformation from the previous into the new meta model and then to adapt the template for the generation of the database schema to this meta model.

**Listing 1.9.** Another example of a meta model based on a relational schema.

```
$schema = new DDLModel('DDL-Demo');
$personTable = $schema->createTable('person');
$att = $personTable->addAttribute('name','String');
...
$att = $filmTable->addForeignKey('director',
                                 $personTable);
$att->notNull();
$att->onDeleteSetNull();
```

The last concept treated is model verification. In analogy to model transformation, it can be implemented in the form of methods operating on the meta model. For example, no isolated classes shall exist within a scheme or each class shall possess a primary key.

*Lessions Learned.*

- Apart from model-to-code transformations, model-to-model transformations are possible. This allows for multi-step transformations within the generator.
- It is distinguished between transformations within the same meta model and transformations between different meta models.
- Model verification is applied to check the model for semantic correctness or completeness before transforming it into code or another model.

The whole picture concerning the generator backend and the generator kernel, as considered in the lecture is shown in figure 1.

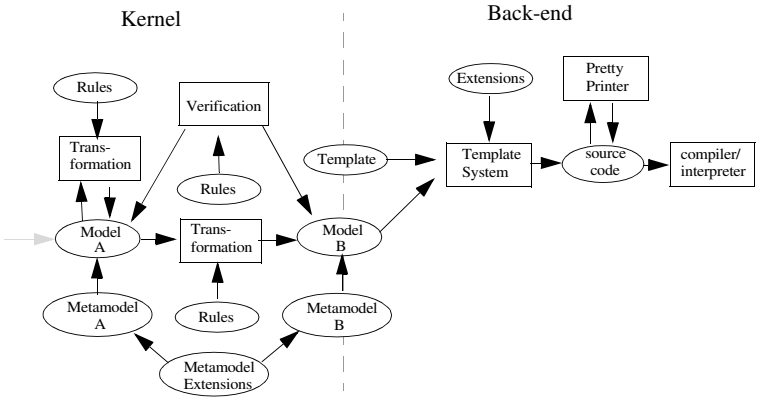


**Fig. 1.** Generator kernel and backend.

**Generator Front-end.** So far, the input of the model has consisted of a series of method calls of the underlying meta model. This is uncomfortable, as the user of the generator is required to have a basic knowledge of the programming language PHP. This is not very elegant and does not allow for a later connection of the graphical modeling tool.

It is therefore reasonable to have an XML-based description of the model and to use it as input for the generator. By using an XML-representation as the modeling language, it later becomes possible to apply a graphical modeling tool.

*Task of the Students.* Generation of an XML language containing all concepts available within the meta model. For this language, a DTD or XML schema has to be developed. Then, an import method must be implemented, by means of which the model formulated in XML is mapped on the methods of the internal meta model of the generator (with the help of PHP DOM API). An example XML model, based on the API model from listing 1.3 is shown in listing **??**:

**Listing 1.10.** Simple xml based language for the generator.

```
<model name="demo">
    <class name="Film">
        <attribute name="title" type="String" length="30"/>
        <attribute name="year" type="int" length="4"/>
    </class>
    <class name="Person">
        <attribute name="name" type="String" length="30"/>
        <attribute name="day_of_birth" type="date"/>
    </class>
    <relation name="is_director">
        <class name="Film" role="film" min="0" max="1"/>
        <class name="person" role="director" min="0" max="*"/>
    </relation>
</model>
```

*Lessons Learned.*

- Formulation of the facts of the internal meta model by a markup language.
- The developed DTD is the meta language of the XML model.
- Implementation of a XML import filter.

In the last step of the first run, a graphical modeling tool shall be connected:

*Task of the Students.* It is the task of the students to analyze the XMI output generated by the modeling tool for all features supported by the generator developed (classes with attributes, various types of relations). Then, XSLT style sheets must be developed, by means of which the XMI format is mapped on the own XML-based model.

As an alternative, it may be done without the transformation into the own XML based meta model and the import filter may directly process XMI. It was found however, that with the intermediate step in first defining a own proprietary XML format it is easier for the students to build the whole import module, due to the rather complex structure of XMI. Another reason for the intermediate step ist that the students have to realize another model transformation (in this case with XSLT) and also have to build an own meta-model (the XML language).

*Lesons Learned.*

- Basic structure of XMI.
- Model transformations (from XMI into the own XML-based format) using XSLT style sheets.
- XSLT style sheets can also be applied to transform within the own XML-based model (e.g. adding a key attribute, if it does not yet exist).
- Discover the disadvantage of using a general purpose transformation language instead of a domain aware transformation language.



With this step, the complete workflow from graphical modeling of the features to be generated to a series of transformation and verification steps, to actual code generation has been implemented. In the subsequent extension step, the generator is extended with additional features in the opposite direction.

### 3.2 Second Run - From Front-end to Back-end

This extension may be an additional UML diagram type, such as the state transition diagram, or the extension of the class diagram by other features (inheritance, methods, additional attributing by stereotypes and tags).

Extension of the generator consists of the following tasks:

1. Analysis as to how the additional UML language elements are expressed in the XMI format.
2. Extension of the own XML language by the additionally needed language elements.
3. Extension of the XSLT style sheets to transform the new features to be supported into the own extended XML language.

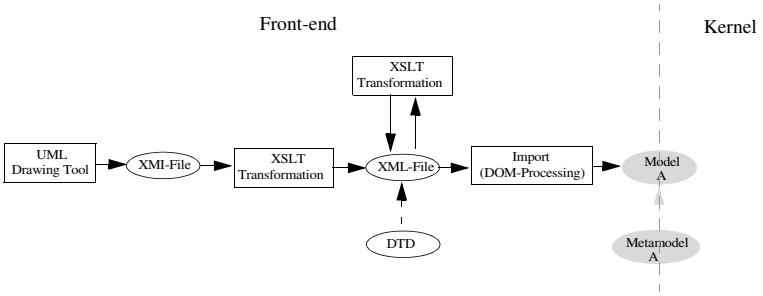


**Fig. 2.** Generator front-end.

4. Extension of the generator-internal meta model.
5. Extension of the import filter.
6. Development of new templates/extension of existing templates to generate the additional code.

The components, composing the generator frontend are shown in figure 2.

### 3.3 Third Run - Implementing a Concrete Task with an Existing Generator Tool

The lecture is then completed by analyzing a freely choosable existing generator tool. It is the objective to handle a simple task by means of the tool and to present this tool to the other students by a presentation of about twenty minutes. Here, the students should always reference the concepts presented in the lecture.

## 4 Summary

The paper describes a didactical model to convey the fundamentals of model-driven software development. In an approach consisting of several steps, the most important concepts in the field of model-driven software development are learned, detailed, and applied in practice. Starting from a simple code generation task at the end of the complete workflow, the task itself and, thus, the generator to be developed are extended constantly. These extensions are the motivation for the introduction of new concepts. After the relevant concepts have been learned and applied in a first run, the generator developed is extended in a subsequent learning step. The central role is assumed by the generator-internal meta model that is implemented in PHP and can be extended easily. The meta model does not only represent the input for code generation proper, but also serves as a starting point for model verification and transformation as well as for XML import.

Finally, the concepts learned are applied to a concrete task using an existing generator/modeling tool. The presentations of the tools used will provide the students with a (small) overview of the generators available on the market.

## References

1. Wall, L., Christiansen, T., Schwartz, R.L.: Programming Perl. O'Reilly & Associates, Inc.(1996)
2. Mark Lutz, M., Romano, R., Read, J. Programming Python O'Reilly & Associates, Inc.(2006)
3. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language O'Reilly & Associates, Inc.(2008)
4. Lerdorf, R., Tatroe, K., MacIntyre, P., Apandi, T.: Programming PHP O'Reilly & Associates, Inc.(2006)
5. Hayder, H.,Maia, J. P., Gheorghe, L.: Smarty PHP Template Programming And Applications Packt Publishing (2006)
6. The Apache Xalan Project http://xalan.apache.org/ (2007)
7. Monte Ohrt, M., Zmievski, A.; Smarty - the compiling PHP template engine http://www.smarty.net/docs.php (2007)
8. Oram, A., Talbott, S.: Managing Projects with make, Second Edition - The Power of GNU make for Building Anything O'Reilly & Associates, Inc.(1991)