Teaching Model-Driven Software Development: Revealing the "Great Miracle" of Code Generation to Students

Andreas Schmidt[1,2] **Daniel Kimmig[2]** **Klaus Bittner[2]** **Markus Dickerhof[2]**

[1] Department of Informatics and Business Information Systems
Karlsruhe University of Applied Sciences
PO Box 2440, 76012 Karlsruhe, Germany
andreas.schmidt@hs-karlsruhe.de

[2] Institute of Applied Computer Sciences
Karlsruhe Institute of Technology
PO Box 3640, 76021 Karlsruhe, Germany
{ daniel.kimmig | klaus.bittner | markus.dickerhof }@kit.edu

Abstract

A didactic approach into teaching model-driven software development (MDSD) is proposed in this paper. The main idea is to focus on conveying underlying concepts, rather than managing a concrete tool or presenting a purely theoretical approach, when teaching MDSD. This objective shall be reached by the development of a simple code generator by the students. For this reason the whole process from graphical modeling to the actual code generation is traversed twice. The first time from back to front to introduce the main concepts of a code generator engine and in a second pass from the beginning to extend the generator by additional functionality. The course 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.

Keywords: Model driven Software Development, Teaching, Practical Approach.

1 Introduction

Interest in model-driven software development (MDSD) has recently increased, due in part to the Model Driven Architecture (MDA) initiative of the Object Management Group (OMG). Consequently, the industry's demand for graduates qualified in this field has grown. It is therefore considered crucial to educate students in MDSD concepts, tools and techniques.

1.1 Model Driven Software Development

The idea behind MDSD concerns designing a model of the system to be developed (for instance, using UML), which is then transformed into code in a specific target language, such as *PHP*, C# or *Java*. Using this technique, there is a strict distinction between the domain-specific and technological aspects of a system. While domain specific concepts are represented in the model (as classes, attributes and methods), technology specific aspects like the target language are determined in the transformation rules, specified by the programmer. Typically the mundane and repetitive parts of the code can be easily generated using this technique, so the programmer can focus on the more complex and challenging parts of a system. Benefits of this approach include higher quality and consistency of code, easier translation to newer or different technologies, as well as shorter time in development (Stahl, Voelter, and Czarnecki 2006).

2 Existing Approaches

In developing a didactic concept for our course, the approaches of other universities that teach in this field were examined. Additionally, the proceedings of the Educators' Symposium (EduSymp; Seidl, M. and Clarke, P., 2010), a major forum for software modeling education, was used as a resource. EduSymp is collocated with the annual ACM/IEEE international conference on Model-Driven-Engineering Languages and Systems (MODELS). The results of studying and interpreting these sources yielded three possible variants of MDSD instruction:

1. *Purely theoretical approach*: An advantage of this approach is that it is focused on the most relevant concepts of MDSD. However, this approach faces the drawback in that students often do not consider it to be engaging. This is the case at universities of applied sciences (UASs), which focus strongly on practical work. Nevertheless, a number of universities were found to be using this approach, although some were partly supported by practical lectures.
2. *Tool-supported approach*: An advantage of this approach is that there is a good initial point for quick initial learning by the student. However, this approach is associated with the risk of the students learning how to use a specific tool, but neglecting to learn the concepts. A case study at EduSymp 2011 (Seidl and Clarke 2011) showed that the Eclipse Modeling Framework (EMF) is

the dominating tool in this field. EMF (Steinberg and Budinski 2009) is an extension of Eclipse. It allows the specification of a structured data model and the subsequent generation of *Java* classes from this model.

3. *Practical approach*: This program-driven approach focuses on conveying underlying concepts rather than the use of a concrete tool. This can be performed by the development of a simple code generator, based on tools that can be commonly used by students. The advantage of this approach is that it is more engaging for the students. One risk with this approach is the danger of excessive workload demand on the students.

3 Surrounding Conditions

The course is integrated in the undergraduate curriculum of a degree program in Business Information Systems at a UAS in Germany. In contrast to traditional universities, UASs have a more applied or practical orientation, and less research (Fachhochschule 2013). As a consequence, they have Bachelor and Masters programs, and not PhD programs. With their strong focus on teaching professional skills, the graduates are oriented for an industry position, rather than academic.

The course should occur at the end of the Bachelor program (6th semester) as a compulsory optional subject of two contact hours per week (2 credits). At this point, the students have already finished courses in Programming (I + II), Modeling, Software Engineering (I + II), Databases (I + II) and Operating Systems. Additionally they have a number of business economics and mathematic based courses. For a complete overview of all courses see (HSKA 2013).

During the last two academic terms of their studies, students can choose from a wide spectrum of different elective courses. The interest of the students for a special subject, as well as the student's availability of time, plays a crucial role in the students' decision to take a course.

Although these students have a solid base in programming skills, it is important to consider that the students are business informatics candidates, rather than computer scientists. Therefore, students have an abstract view of software generation before enrolling in this course, as depicted in Figure 1.

Figure 1: Broad idea of a code-generator

4 Our Approach

From the three possible variants of teaching MDSD, as previously articulated, the pure theoretical approach is eliminated. The reasons for this were as follows:

1. This approach often is not seen as engaging by students. Therefore, in order to compete with courses in subjects, such as mobile business or social media (which are very popular at this time), it is unlikely that this this approach is perceived as attractive by students.
2. Additionally, students at UASs have a stronger focus on practical work, therefore a theoretical approach would seem to be less appropriate in this learning environment with an applied focus.

Because the construction of a code generator could be a complex task, it was decided to use an existing code generator in our course. With this approach, students could get an initial feeling of success quite early in the course. For that reason, a number of different code generators were examined, but none were deemed satisfactory, the main reason being that many of the available generators are very complex or specialized, and not easy to use. Therefore, there is a danger that students will only learn to handle a particular tool, and the concepts of MDSD will have less emphasis. Due to its wide use in MDSD-related courses, the Eclipse Modeling Framework (EMF) was examined, yet rejected. The lack of dedicated tools for teaching MDSD was also one of the big challenges formulated in the position paper about Software Modeling Education from Seidl and Clarke (2011).

Parallel to the evaluation of the different code generator tools, the topics and central concepts of the course were selected. A number of core topics were identified, as well as a number of optional topics considered as "nice to teach", depending on time constraints. The core topics comprise of:

- Code generator types
- Models
- Meta-models
- Model transformation
- Model verification
- Template systems
- XML Metadata Interchange (XMI)

Domain specific languages (DSL) are disregarded, due to time constraints (Fowler 2010).

Considering the slightly-exaggerated initial view of a code-generator held by students (See Figure 1, above), one of our main efforts would be to remove the confusing area of the "miracle cloud" in the middle. Using an existing code generator tool like EMF leads to the danger that this cloud will remain, because the "miracle" happens inside the tool. To address this, a third approach in teaching MDSD (the practical approach) was considered, which involves letting students build a generator by themselves. This should lead to students to understand the whole process of code generation. In this case, no "miracle cloud" will remain because students will gain insight as to how code is generated. This insight will aid students in their future use of various code generators, or construction of small or medium size generators for specific problems.

The development of the code generator is accomplished by a number of consecutive exercises. In order to reduce the complexity, the capability of the generator will be limited to the generation of a subset of UML class diagram capabilities (classes, attributes, and relations). Additionally, the initial meta-model will be provided in the form of an extensible library. In this way,

the complete workflow starting from graphical modeling to actual code generation is conveyed.

4.1 Tools and Technologies

A careful selection of appropriate tools to support the implementation of the code generator has to be done. Because there are only two contact hours a week with students, tools and languages are selected because they are largely known by the students, and also because an excessive learning expenditure is not required to learn the tool. For instance, the use of a descriptive language, such as the Object Constraint Language (OCL) for implementing constraints (Warmer and Kleppe 2003), is disclaimed. Instead, the constraints are formulated in a procedural manner. At this point, it will also be referred to the fact that other often more comfortable, but also more complex solutions exist. Understanding the generator principles however is not limited by these simplifications. It enables the student to concentrate on the most important points.

4.1.1 Programming Language

The code generator should be implemented using a scripting language, due to its generally high accessibility and prevalence, and good string handling. Possible candidates are *Perl* (Wall, Christiansen, and Schwartz 1996), *Python* (Lutz 2006), *Ruby* (Flanagan and Matsumoto 2008), and *PHP* (Lerdorf, Tatroe, MacIntyre, and Apandi 2006). The language *PHP* was selected, due to the fact that our students have already used this language at this point in time, and that *PHP* is a so-called macro language, meaning that it already supplies a template mechanism.

4.1.2 Meta Model

To accelerate learning, a small *PHP* library is supplied to the students, which represents the initial meta model and allows the formulation of their models by an API. The meta-model allows the formulation of classes, attributes, and relations. In addition, the meta model supplies a number of methods or properties to iterate 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.

4.1.3 Template Engine

The template mechanism incorporated in *PHP* is used at the beginning of the course, but later on replaced by the explicitly available template engine *Smarty* (Hayder, Maia, and Gheorghe 2006), a widely-used module in the PHP community, which is considered to be highly mature. As well, Smarty has high quality documentation (Ohrt and Zmievski 2012).

4.1.4 UML-Modeling Tool

The only requirement made on the modeling tool is that an XMI export format can be written. This is provided by most modeling tools.

4.1.5 XML

The external model to be developed and the XMI model generated by the UML-modeling tool are both XML-based. For this reason, DOM, XSLT, XQuery and XPath functionalities are needed, which are provided from *PHP* by default. *Xalan* (Apache 2007) and *Zorba* (Zorba 2013) are used as external XSLT and XQuery transformation tools.

4.1.6 Workflow

Many steps are required to get from graphical modeling over various model transformations to the actual code generation. Often, certain parts of code generation (mostly transformations) are accomplished within the framework of development. Consequently, these parts shall not be delivered in a monolithic block, but exist as external components with clearly defined interfaces, which are then assembled by the UNIX Tool *make* (Oram and Talbott 1991).

4.2 Didactic Approach

Code generators are divided into three parts: generator-frontend (definition of model); generator-kernel (model transformation, verification, etc.); and generator-backend (code generation). A multi-stage approach was selected. In the first stage, the process is started at the back end (code generation), and worked successively to the front end. 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 covering code generation. At the end of the first stage, a complete execution chain exists from modeling with an external UML modeling tool to several transformation steps, to the generation and formatting of the source code. The second stage is aimed at extending the code generator with additional capabilities, such as inheritance or supporting other UML diagram types, such as state transition diagrams. Contrary to the first stage, it proceeds in the other direction. In other words, the extensions are made starting from the output of the UML modeling tool to the meta model, to code generation by templates.

Figure 2 gives an overview of the exercises in the first two stages. Due to this double treatment of basic components of a code generator, in-depth understanding of the functioning of a generator is obtained, which exceeds the understanding obtained from learning a concrete tool or from the purely theoretical approach.

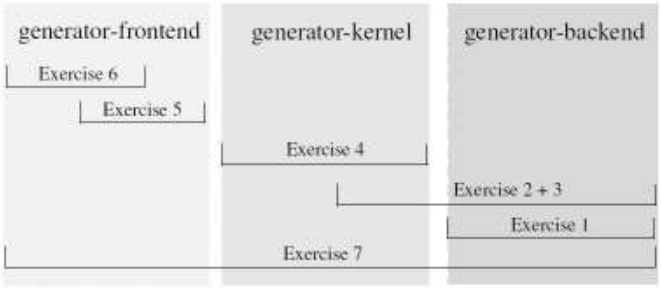

Figure 2: Code generator parts and exercise coverage

The course 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.

5 Exercises

A number of consecutive exercises are assigned to students of the course. Within the framework of these exercises and the accompanying theoretical lessons, the following concepts will be covered: templates, model, meta-model, model verification, transformation between the same as well as between different meta-models, XML-based models and meta-models, XMI.

5.1 First Stage - From Backend to Frontend

There are six exercises in the first stage. They are named 1) generating simple *Java* classes, 2) Supporting relationships, 3) use of the new Meta-model and explicit template engine, 4) model verification and transformation, 5) using an XML based model and 6) connecting a graphical UML modelling tool. These exercises are described in detail below.

5.1.1 First Exercise - Generating Simple Java Classes.

The starting point is the actual code generation. For this purpose, a simple *Java* source code consisting of some classes is presented to the students. Figure 3 shows a single class that should be generated. Then, the students are asked to implement a minimum PHP program that generates the given *Java* source code.

```
class Person {

    protected String name;
    protected Date yearOfBirth;

    public Person() {}  //empty constructor

     String getName() {
       return name;
    }

    void setName(String value) {
       name = value;
    }

    Date getYearOfBirth() {
       return yearOfBirth;
    }

    void setYearOfBirth(Date value) {
       yearOfBirth = value;
    }
}
```

Figure 3: Single Java class to be generated

It is now the task of the students to analyse the source code for parts that represent recurrent concepts for all classes and, hence, can be generalized and parts that are class-specific. In doing this, the students separate the model from the template part of a generator. Additionally, they implicitly define a simple meta-model for their model, consisting of an appropriate data structure. Lastly, they use the PHP macro feature as an example of a template system. Figure 4 shows a simplified result to be produced by the students.

5.1.2 Second Exercise - Supporting Relationships

In the next lesson, the model shall be extended to cover relationships (1:n, n:m) among classes. The first task the students have to perform is extending the syntax of the model. Most of the students' solutions support predefined data types, as well as class names and lists, or a combination of both (i.e.

`'director'=>'Person:directs',`
`'actors'=>'List(Person):plays'`).

The implementation of the semantics has to also be done. This is realized in the template part. Here, the templates tend to get a little bit cluttered. At this stage students generally realize that their model includes a number of weak points. Firstly, the model is completely unrestricted and secondly it is error-prone, in the case that wrong information is provided.

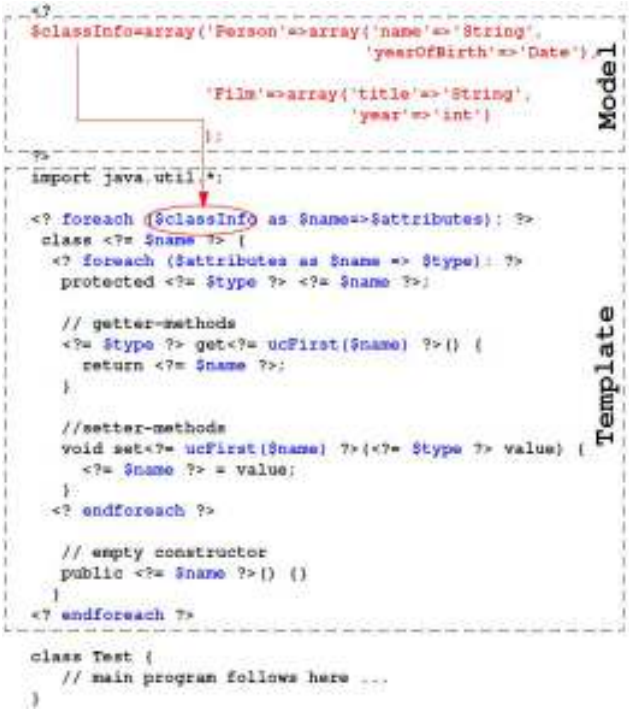

Figure 4: Simple, hardcoded code generator for the generation of two Java classes

As an extension of this exercise, students are asked to generate DDL code for a relational database. At first sight this looks not more difficult than generating the *Java* code, but ends up in an unpleasant surprise. The fact that the relationships are implemented with foreign keys does not allow the generator to sequentially access the model information and transform the current element into the target code. Such an operation requires random access to the model schema to obtain the information about the corresponding primary key and its data type.

These problems will be used as motivation to work on an improved meta-model, which is then presented to the students. The meta-model allows the definition of the model via a number of API calls. The meta-model also allows accessing the information in the model by navigating along properties, and has an extension mechanism to meet special needs for the different target languages. A basic example is given in Figure 5, illustrating the use of the API. In the upper part of the figure, the definition of a model (Name: Demo) with two classes and a relationship between them is shown. In the lower part, the "how to iterate" over the model information is illustrated. In this example, the information about the properties and the primary key of each class is

printed. The corresponding ER-model is shown in Figure 6.

Model definition

```
// Definition of model:
//
$model = new Model('Demo');

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

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

$film->addRelationEnd('director', $person,
                      '<0,1>','directs');
$person->addRelationEnd('films',$film,
                        '<0,*>','directs');
```

Iteration

```
// Iteration over the attributes of a class:
//
foreach ($model->classes as $class) {
  $pk = $class->primaryKey;
  print "Class: $class->name ($pk->name)\n";
  foreach ($class->attributes as $att) {
    print "$att->name $att->type->name\n";
  }
}
```

Figure 5: Definition of the model from Figure 6, using the Metamodel-API

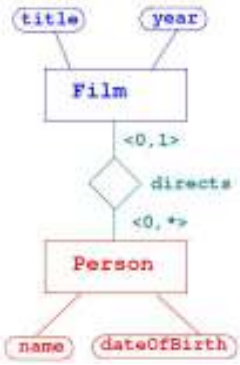

Figure 6: Simple model

5.1.3 Third Exercise - Use of the new Meta-model and explicit template engine.

When generating the database schema in the last exercise, because of the required non-linear access to the model, the templates often gets "contaminated" with snippets of PHP code, which made the template difficult to read and maintain. In order to address this topic, a simple solution is presented to our students. When utilizing an explicit template engine (*Smarty*), cluttering the templates with *PHP* code snippets is not possible. A template system has a small number of specialized language elements, which are well suited for generating arbitrary output, but not to solve general programming problems. Tasks for providing the required template data can be provided by extending the given meta-model or by extending the template system by user defined functions and so called 'modificators'. These techniques keep the templates small and readable. For that reason, in this exercise, students have to build the model with the new provided API-based meta-model and also to adapt their existing templates from the PHP-macro functionality to *Smarty*. Additionally, students have to overcome the problem of platform-specific data types for *Java* and the DDL code. To facilitate the implementation of the templates for the different targets, students also extend the provided meta-model with appropriate methods, which are mostly specific to the target platform. Figure 8 shows a possible solution developed by the students. The methods `is1N()`, `is11()`, `getCard1Side()` and `getCardNSide()` are extensions of the provided meta-model implemented by students. Additionally, for solving the problem with the different datatypes in Java and *MySQL*, the students have to extend the functionality of the Smarty template engine with appropriate modifiers (i.e. `mysql_type`).

5.1.4 Fourth Exercise - Model Verification/ Transformation.

In this exercise, a number of tasks have to be performed on the meta-model. Firstly, some verifications of the model must be implemented. Secondly, each class in the model is extended by some administrative fields (which demonstrates a model to model transformation on the same meta-model). Finally, a transformation to another, more platform-specific meta-model (with concepts such as table, foreign key, relationship table, etc) must be implemented. Figure 7 shows an example of a basic model transformation within the same meta-model, by adding two administrative fields to each class.

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

Figure 7: Simple Model to Model Transformation

5.1.5 Fifth Exercise - Using an XML based Model

In this exercise an XML-based language, which represents the concepts in the meta-model, has to be developed by the student. Next, an import filter that maps the developed language to the metamodel API must be implemented.

5.1.6 Sixth Exercise - Connecting a Graphical UML Modeling Tool

In the sixth and last exercise in the first stage, a graphical modeling tool is connected. This has a number of steps. First, the XMI language has to be analysed. Afterwards an XSLT or XQuery transformation from XMI to the previously developed XML language must be carried out. Figure 9 shows an example of a simple XML-language, covering the model concepts. With this step, the complete workflow from graphical modelling 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 (second stage), the generator is extended with additional features in the opposite direction.

```
[@ foreach from=$model->relations item=relation @]
  -- Relation [@ $relation->name @]
  [@ if $relation->is1N()==1 @]
    [@ assign var=end1 value=$relation->getCard1Side() @]
    [@ assign var=endN value=$relation->getCardNSide()  @]
     alter table [@ $end1->class->name @]
       add [@ $end1->role @]_fk [@ $endN->class->primary_key->type->name|mysql_type @]
       references [@ $endN->class->name @]([@ $endN->class->primary_key->name @]);
  [@ elseif $relation->is11()==1 @]
    -- 1:1 relationship
  [@ else @]
    -- n:m relationship
  [@ /if @]
[@ /foreach @]
```

Figure 8:Template for the 1:n relationships (DDL-Code)

5.2 Second Stage - From Frontend to Backend

The exercise in the second stage consists of extending the functionality of the code generator by additional features, such as another UML diagram type (i.e. 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.

First, the students have to analyse how the newly supported UML language elements are expressed in the XMI format. Then, they have to extend their own XML language by the additionally needed language elements.

```
<?xml version="1.0"?>
<!DOCTYPE my-model SYSTEM "my-meta-model.dtd">
<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="directs">
        <class name="Film" role="film"
               min="0" max="1"/>
        <class name="person" role="director"
               min="0" max="*"/>
    </relation>
    <relation name="has_role">
        <class name="Film" role="film"
               min="0" max="*"/>
        <class name="person" role="actor"
               min="0" max="*"/>
    </relation>
</model>
```

Figure 9: Example XML-based model

Afterwards, the XSLT/XQuery style-sheets must be adapted to support this transformation as well. This extension then continues over the generator-internal meta-model, the import filter, and ends with the development of new templates to generate the additional code. Figure 10 shows the coverage of the individual exercises inside the whole workflow of the generation process. It can be clearly seen that this approach starts from the backend of the generator.

5.3 Third Stage - Implementing a Concrete Task with an Existing Generator Tool

The course is then completed by analysing an existing code generator tool, which is freely available and chosen by the student. The objective is to handle a simple code translation task with the tool, and to present this tool to other students in a presentation of about twenty minutes in length. In this presentation, all students should refer to the concepts presented in the course.

6 Evaluation of our Approach

The course has been offered once a year since 2006. After students present results of an existing code generator or related tool/technology at the end of the course, they are asked for their impression about the didactical concept. Interestingly, even the students who at the beginning of the course would have preferred the use of an existing tool (mostly, because they already have some experience in MDSD) rather than building a code generator itself, changed their mind and favor the chosen approach. The insight they received from building a small but complete code generator from scratch by far exceeds their prior understanding of such tools. Further, students who had no previous experience with MDSD stated that building the code generator helped them in understanding the chosen generator at the end of the course, as knowledge of the internals of a code generator was acquired at that time. Additionally, students feel prepared to be able to develop small or medium size code generators in the future. Even so, some students are intimidated when they find out that they have to build a generator in this course, prompting them to choose another course.

An empirical study from Whittle and Hutchinson (2011) investigated how industry uses MDSD produced findings relevant to the instruction of MDSD. First, a lot of MDSD examples concern small generators and DSLs developed in as little as two weeks. Another interesting finding of the study was that successful MDSD practice starts from the ground and doesn't follow a heavyweight top-down approach. Top-down means that students have to develop abstract models first, refine them into architecture, and then finally to code. Formulating abstractions of a system before the details are full

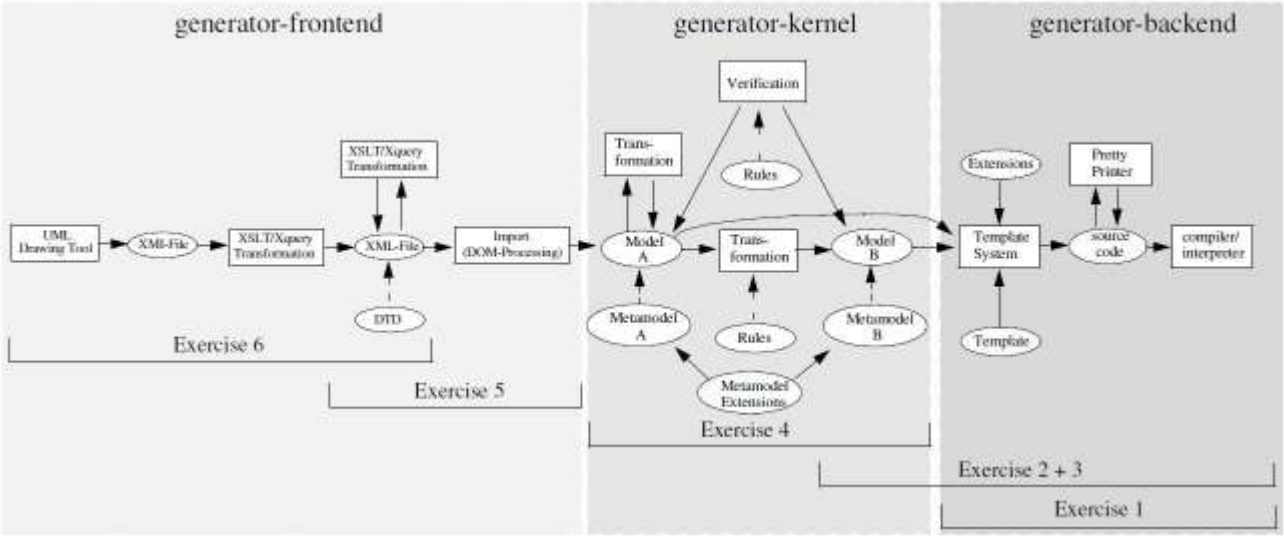

Figure 10: Coverage of the different exercises (phase one)

understood is quite a difficult job, compared to the bottom-up approach with starts for looking after reusable assets in the code and then creating abstractions from them. This can be compared with the approach described in this article, starting at the backend of a generator, leading the student to understand the necessity of components introduced as the course progresses. With a start at the beginning (XMI), there would have been much more problems to motivate further steps.

7 Summary and Outlook

The paper describes a didactical model to convey the fundamentals of MDSD in the classroom environment. In an approach consisting of several stages, the most important concepts in the field of MDSD are learned, detailed, and applied in practice. As of 2013, the course has been lectured 8 times. In the evaluations, students mention a higher workload than average, compared to other elective courses, but nevertheless rank it as one of their favorite courses. In particular, the first stage starting from the backend and motivating the need of further MDSD-specific concepts is highly appreciated, which can be documented by means of the very high rating for the central theme.

While the actual platform for implementing the code generator is *PHP*, *Smarty* and *make*, it is planned to also provide some exposure to the Java framework (using *ant* and *freemarker*) for the future. This framework should not replace the existing *PHP* solution but should be an alternative for students who prefer *Java* instead of *PHP*.

Another approach currently under development is providing students with a code generator skeleton which represents the state after the first stage as a starting point, and continues by extending the generator with additional functionality.

8 Acknowledgement

We like to thank Michael Dohan from Lakehead University in Thunder Bay, Ontario, Canada, for proofreading of paper, and translating it into scientific English – thanks Michael!

9 References

Apache (2007): The Apache Xalan Project http://xalan.apache.org/. Accessed 25.10.2013

Fachhochschule (2013): http://en.wikipedia.org/wiki/Fachhochschule. Accessed 25.10.2013

Flanagan, D., Matsumoto, Y. (2008): The Ruby Programming Language O'Reilly & Associates, Inc.

Fowler, M. (2010): Domain Specific Languages (1st ed.). Addison-Wesley Professional.

Hayder, H., Maia, J. P., and Gheorghe, L (2006): Smarty PHP Template Programming and Applications, Packt Publishing.

HSKA (2013): Bachelor Degree in Business Information Systems, Karlsruhe University of Applied Sciences. http://www.hs-karlsruhe.de/en/faculties/computer-science-business-information-systems/business-information-systems-bachelor.html, Accessed 25.10.2013

Lerdorf, R., Tatroe, K., MacIntyre, P., Apandi, T. (2006): Programming PHP O'Reilly & Associates, Inc.

Lutz, M. (2006): Programming Python O'Reilly & Associates, Inc.

Ohrt, M., Zmievski, A. (2012): Smarty - the compiling PHP template engine http://www.smarty.net/docs/en/. Accessed 25.10.2013

Oram, A., Talbott, S. (1991): Managing Projects with make, Second Edition - The Power of GNU make for Building Anything O'Reilly & Associates, Inc.

Seidl, M., Clarke, P. (2010): Software Modeling in Education: Proc. (Educators Symposium) at 13th ACM/IEEE International Conference on Model-Driven Engineering, Languages, and Systems, Oslo, Norway.

Seidl, M., Clarke, P. (2011): Position paper: Software Modelling Education. The 7th Educators Symposium at Models, Wellington, New Zealand

Stahl, T., Voelter, M., Czarnecki, K.. (2006): Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.

Steinberg, D., Budinsky, F., Paternostro, and Merks, E. (2009): Emf: Eclipse Modeling Framework 2.0 (2nd ed.). Addison-Wesley Professional

Wall, L., Christiansen, T., and Schwartz, R.L. (1996): Programming Perl. O'Reilly & Associates,Inc.

Warmer, J. and Kleppe, A. (2003): The Object Constraint Language: Getting Your Models Ready for. Addison-Wesley Longman Publishing Co., Inc.

Whittle, J. and Hutchinson, J. (2011): Mismatches between industry practice and teaching of model-driven software development. In Proceedings of the 2011th international conference on Models in Software Engineering (MODELS'11), Jörg Kienzle (Ed.). Springer-Verlag, Berlin, Heidelberg, 40-47.

Zorba (2013): http://www.zorba-xquery.com/. Accessed 25.10.2013