Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1647–1656
DOI: 10.15439/2014F266
ACSIS, Vol. 2

Pragmatic Model-Driven Software Development from the Viewpoint of a Programmer: Teaching Experience

Jaroslav Porubän, Michaela Bačíková, Sergej Chodarev and Milan Nosáľ
Technical University of Košice, Department of Computers and Informatics
Letná 9, Košice, Slovak Republic
Email: {jaroslav.poruban, michaela.bacikova, sergej.chodarev}@tuke.sk, milan.nosal@gmail.com

***Abstract*—Model-driven software development is surrounded by numerous myths and misunderstandings that hamper its adoption. We have designed our course of model-driven development approach with the goal to introduce it from the viewpoint of a programmer as a pragmatic tool for solving concrete problems in development process. The course covers several techniques and principles of model-driven development instead of concentrating on a single tool. To explain these techniques we use a case-study that is iteratively developed by the students during the course. In the paper we explain the structure of our case study, contents of individual iterations, and our overall experience with this approach.**

I. Introduction

MODEL-DRIVEN software development approach (MDSD) promises increase of development speed and quality of resulting software by the use of formal model of the system as a basis for its implementation [1]. Understanding MDSD, however, suffers from several myths that hamper its adoption. This section discusses these myths to provide the motivational context to our work. In the rest of the paper we present our approach to MDSD teaching that is tailored to overcome these myths.

A. Myth 1: MDSD is a large-scale approach

MDSD is mostly viewed from the perspective of large-scale software architecture. A new system or a family of systems is supposed to be implemented by describing every significant part and aspect of the system using formal models (for example in [2]). In practice, however, it is not always the case. When a system development begins, it may not be known that a whole product family would be needed in the future. Therefore, it is not clear beforehand that model-driven development would be applicable and that investment in it would pay off.

Of course, in reality MDSD can be considered in a smaller scale, where only specific parts of the system are generated based on models. In this case, the knowledge of MDSD can be useful even for a single programmer (or a small team) working on a part of the system and introduction of MDSD may not require significant changes in the architecture of the system as a whole.

B. Myth 2: MDSD requires massive tool support

MDSD is often associated with integrated modelling tools or language workbenches. These tools cover development of meta-model, a domain-specific language used to express models and a generator that produces runnable code based on a model. Modelling tools may also provide environment for development of the model itself. These tools, however, are often complex and require high learning costs. What is more important, the use of such tools poses the risk of vendor lock-in.

Although integrated tools may be useful in a lot of situations, they are not necessarily required by the model-driven approach. It is possible to use a set of independent tools for separate parts of the model-driven development infrastructure (e.g., for language processing, for code generation, etc.). This approach allows looser coupling and greater flexibility in the choice of tools.

C. Myth 3: MDSD requires special software development process

It is considered that model-driven approach requires the use of a special software development process, where meta-model and modelling language must be completely specified and implemented before a model of a system can be developed. This opinion renders MDSD as very inflexible and incompatible with agile development processes that are currently favoured.

Modelling infrastructure, however, can be developed iteratively. Meta-model, language processor and generator can evolve together with the rest of the system. The use of small-scale MDSD and simpler tools as described in previous paragraphs greatly simplifies such iterative development process and allows using MDSD along with common agile methodologies.

D. Myth 4: MDSD is not widely used in practice

Without a deeper insight it seems that MDSD is not a widely used approach in practice. In reality, model-driven and generative approaches are indeed wide-spread and even considered a good practice for pragmatic programming [3]. Most of the examples, however, represent small-scale MDSD applications which include:

978-83-60810-58-3/$25.00

- Generators of database schema and object-relational mapping (ORM) code from the description of a data structure (used in various ORM tools).
- Generators of code for accessing web services based on WSDL description.
- Tools for graphical user interfaces design that generate code according to a graphical representation of the user interface.
- IDE plugins for specific technologies that are able to generate skeletons of repetitive artefacts (e.g., GWT plugin for IntelliJ Idea that can generate standard GWT RPC service artefacts).
- Spring Roo generative framework that allows to implement custom code generators for various repetitive code artefacts (currently published generators focus on web-based CRUD application domain).

Furthermore, the MDSD application is often hidden from a programmer by libraries and frameworks that allow to specify behaviour using a model without knowing details of model processing. In case of dynamic languages such as Ruby, internal domain-specific languages can be used for description of models and code generation can be replaced with run-time program modification using reflection. This approach makes the use of MDSD even less obvious.

II. Pragmatic Model-Driven Programming

We designed the MDSD course with these considerations in mind. The course is intended for graduate students that would mostly become software engineers in their future career. Because of this, we wanted to demonstrate the approach from the viewpoint of a programmer. This led us to the following goals:

1) *Keep it practical.* We wanted to maximize the possibility that our students would be able to use the learned skills and techniques in their future careers. This means that these techniques should be applicable in a wide range of situations.
2) *Teach principles using realistic examples.* Students should understand the basic principles of the topic as they have much higher level of applicability than any concrete tool. At the same time we should illustrate these principles using realistic examples, tools and approaches that can be directly used in practice.

For these reasons we need to teach MDSD in a way that challenges myths described in the previous section. First of all, we show the students that model-driven approach is not limited to large-scale solutions. Although we demonstrate development of a complete system using MDSD, parts of the system are modelled and generated separately showing different scales of modelling.

We also do not use any full-fledged MDSD tool for the whole development process. Instead, we concentrate on several development techniques and tools behind them from the perspective of the main components of the MDSD infrastructure. If we apply the language-oriented perspective, we can divide the components and techniques that we teach as follows:

1) *Abstract syntax* – meta-model that describes structure of models:
   - definition of meta-model using classes in object-oriented language,
   - composition and reuse of models.
2) *Concrete syntax* – domain-specific language used to specify a model:
   - implementation of delimiter-directed parser,
   - internal domain-specific languages,
   - the use of parser generator,
   - generic XML parser.
3) *Semantics* – generator used to produce code based on the model:
   - generation by direct transformation,
   - generation using templates, templates composition and reuse.

All of the listed technologies and approaches are used in the course. In the beginning of the course students are building simple version of MDSD tools themselves (e.g. parser, generator). Later on, when the complexity of tasks arises we are fluently switching to the well-known MDSD tools (e.g. parser generators, tempalting engines). In the end of the course, students are able to compare tools and approaches and choose the most effective one in a particular situation.

The techniques that we teach can be combined in various ways to power MDSD. At the same time, they can be used even separately for a wide range of programming tasks. From this aspect our approach is similar to the one used by Folwer in [4].

All taught techniques are demonstrated on a single case study that is developed in an iterative manner. Meta-model used in a case study is gradually extended showing evolution and composition of meta-models and languages during the system development. We also show that a meta-model can be reused even if the implementation of a language processor is replaced. Thus we demonstrate parallel evolution of different components of the MDSD infrastructure. It also shows the possibility to use usual agile development processes with connection to MDSD.

III. Case study

In this section we introduce the case study we use to teach MDSD in our course. *CrudComp* is a fictive company, which develops CRUD (create, retrieve, update delete) applications for different domains storing entities and their properties and relationships (e.g. employee, department). They have started with a single application but their success brought them new customers. *CrudComp* soon identified fundamental requirements shared across all the applications they developed for their customers. Each CRUD application has to provide means for:

- data entry,
- data validation,
- data persistence into an (external) memory, and
- data presentation in a user interface.

They also identified a few non-functional requirements for the applications concentrating on technology.

- *User interface technology* – the customers expect a migration to a new user interface type in the near future (e.g., mobile, web).
- *Storage technology* – currently the applications are supposed to work with a relational database, but some of the customers consider transition to a file system or a NoSQL database.
- *Service-oriented architecture* – *CrudComp* providentially expects that the customers will want to export the CRUD functionality as the web services for integration with other business applications.

These specifications led the *CrudComp* to design a multi-tier architecture for their CRUD applications consisting of 3 layers: user interface, service and data access. The architecture skeleton is depicted in Figure 1.

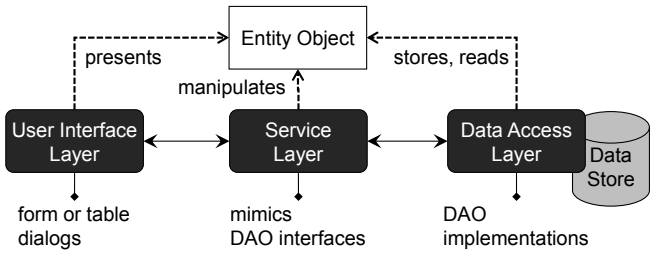

Fig. 1. Multi-tier architecture used by *CrudComp* CRUD applications

The architectural framework defined applications skeleton, but there would still be quite a lot of repetitive work. Luckily, one of the *CrudComp* employees was an enthusiast for MDSD and generative programming. The vision presented to the management was that the MDSD solution would not only speed up the development, but it would also make the communication with the customer more efficient. The domain-specific language used to specify the model of an application would be readable also by customers (in this context they were domain experts). Moreover, the customer would be even able to *write* the input file by himself. This way *CrudComp* would not only spare the time and money for the development, but it would also gain more effective ways to communicate with the customer. They have started with the application of the MDSD in a small scale (generating just small parts of the system) not to hamper the development of currently developed applications. And so he with the rest of the team decided to iteratively build a full MDSD solution with each iteration covering just a small portion of the whole CRUD application. Until the whole solution would be ready the programmers would have to implement the rest of the system manually to meet the deadlines.

The input of *CrudComp* MDSD solution was a simple model of the domain in terms of entities, their properties and relations. The output was supposed to be a part of a CRUD application. They decided to start with entity classes generation since the entity classes are the simplest artefacts of a CRUD application source code. Entity classes define the data structures manipulated in a CRUD application. Later they wanted to generate also data-access objects implementations, validators, etc. The only thing the programmers would need to do to get entity classes or later even the complete CRUD application is to write a simple input file that models the domain for a given customer. After running the generator they only needed to add a hand-written specific code to the generated source code to finish the application so that it would be fully functional and tailored for the specific customer. Thus they were able to reduce the amount of their repetitive work.

CrudComp presents a simple case of possible MDSD adoption candidate. As the reader can see we use the case study to motivate and explain the adoption of the MDSD technique. However, on the other hand we do not avoid development process issues. The case study context puts students in the role of *CrudComp* developers that need to iteratively and incrementally build an MDSD solution for a CRUD application family. Incremental MDSD adoption is a necessity to fit into the agile development processes.

We selected a CRUD application software family for our case study because it is the most common information system type in practice. It is also very similar to common project assignments at our university that our students are already used to. Moreover, examples in many tutorials (e.g. Spring, NetBeans JSF) are illustrated on a CRUD application.

During the course, each student works individually on his/her *CrudComp* project at home or in the class. Their progress is controlled in the class on a week basis to prevent procrastination and to help them with issues that raise during the development of the case study.

IV. Teaching MDSD Iteratively

The *CrudComp* case study is divided into 4 relatively self-contained incremental iterations. They all solve problems in the same problem domain – the implementation of CRUD applications. However, rather than solving the whole problem at once, each iteration solves a smaller sub-problem of the CRUD application generation. The first iteration starts with generating entity classes and DAO implementations, and each next iteration adds generation of some new functionality. This iterative approach allows us to teach 4 different approaches to MDSD while keeping the case study quite simple[1]. Thanks to the variety of the MDSD techniques used in the case study we can give the students a brief insight into the problems, advantages and disadvantages of these techniques and the students can compare them by themselves. Multiple approaches also enable us to introduce the problem of language composition.

While each iteration uses a different technique they all share the same tooling infrastructure used in MDSD (see Figure 2).

[1]Of course, this iterative approach is not only about showing multiple techniques. As we argued in Section II, its main objective is to explain (and illustrate, too) to the students that MDSD can be applied just to small portions of the whole system and one can even combine multiple approaches in context of the same system. Moreover, iterative approach nicely fits into agile development.

As the reader can see from the scheme in Figure 2 we accent the importance of the model that connects the problem domain with the implementation.

The case study is divided into 4 iterations that introduce the following techniques:

1) *Entities DSL* processed by a simple parser implemented in an ad hoc manner (delimiter-directed parser). This iteration solves the entities definition problem and enables the user of the generative system to generate the data tier for the CRUD application. Source code artefacts are generated by direct transformation and using templates.
2) *Constraints DSL* implemented as a Java-based internal language. It solves the problem of defining constraints upon entity properties and introduces the technique of language composition. Templates composition is introduced.
3) *Entities DSL with references* processed by a generated parser. The language adds a new functionality to the generated CRUD applications that allows to specify relations between entities.
4) And finally, *UI specification language* parsed by a standard XML parser. The UI specification language enables *CrudComp* to generate a standard user interface for a CRUD application. This iteration introduces templates reuse.

Each iteration ends with a full MDSD solution to a subproblem of the whole CRUD application generation. They all follow the whole scheme in Figure 2. The students can see that MDSD can be applied in a small scale and that it can be done relatively easily and quickly (they can see that even MDSD can be done in agile manner). The following sections discuss the iterations in detail and explain our motivation for each of the chosen approach.

A. Entities Language

The first iteration starts with a simple external DSL. From the viewpoint of the parsing approaches the objective of this iteration is to show the students that writing an ad hoc delimiter-directed parser for a very simple language can be the right choice – in some simple situations the "big guns" such as parser generators could just complicate the matter. To make the implementation of the entities language as simple as possible we exploit the file system for the concrete syntax (see Section IV-A2). On the other hand, the students can also see that if the language would get a little bit more complex, the parser implementation complexity could raise much more (thus we are preparing the ground for introducing other approaches).

Another reason why we start with an ad hoc parser is that students are often scared of parser generators. Usually they think parser generators are complicated and therefore can be used only by experts in language theory. We start with a simple ad hoc implementation to gain the students' attention and enthusiasm.

From the viewpoint of code generation we use both template-based generation and direct transformation. Again, we want the students to understand when a direct transformation generation is enough and when we can simplify generation with templates. To emphasize incorporating multiple approaches in one project we also generate multiple outputs from the same model.

1) Abstract Syntax: In the first iteration we start with a domain model that considers entities and their properties as the data structures handled by CRUD operations. Entity has a unique name and a set of its properties. Each property has a name too (that is unique in scope of one entity) and a type. For the purposes of the project string, integer and floating point number types suffice. The result of the first iteration is a language that covers the domain of CRUD entities.

The abstract form of a language sentence is represented by an in-memory object-oriented model – semantic model using Fowler's words [4]. For each entity there should be an in-memory object that would have a reference to a string with an entity name and a reference to a list of objects representing its properties, etc. The in-memory model is defined by GPL classes (Java classes in our case study). The language model for the problem domain of this iteration is shown in Figure 3.

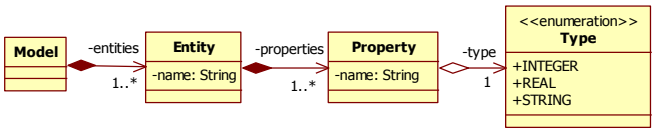

Fig. 3. Object-oriented language model of the entities language

2) Concrete Syntax: The first parsing approach we want the students to use is an ad hoc delimiter-directed parser. To keep the implementation effort manageable we chose a simple pragmatic syntax. The whole model is defined in a single directory in a file system. In this model directory there is a set of files that define entities. One file specifies one entity. The name of the entity is derived from the name of the file. This way we mimic the Java programming language that requires the name of the file to be the same as the name of the class specified by that file. Since we use standard file system to specify the set of entities in the domain model we can keep the internal structure of the entity files simple. The internal structure of the entity files is used to specify entity properties, each on a single line. An example of a language sentence is shown in Listing 1.

Listing 1. Two entities specified in a file-based entities language

```
<model>
  |———— <Department>
  |            name : string
  |            code : string
  |———— <Employee>
               name : string
               age : integer
```

The students have to implement a simple parser `LineParser` that scans a given directory for files and parses them to create in-memory objects representing entities. File system scanning is done using standard Java File API. Files are

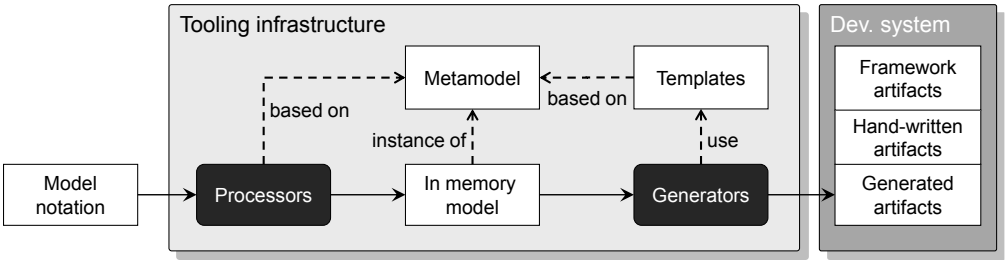

Fig. 2. Tooling infrastructure of our MDSD case study solution

text based and are parsed using just the `String` class and its methods and regular expressions. The syntax of an entity file is very simple. Each line in the file specifies a single property of the entity. The property definition starts with property name and ends with its type that is separated from the property name by a ':' character. Entity files support single line comments starting with '#'.

3) Semantics: Once the students have the in-memory model they use it to generate source code artefacts. In the first iteration they generate just a part of the whole system – the data tier of the CRUD application (simulating small-scale MDSD). For example, for the Employee entity from Listing 1 they are supposed to generate a Java entity class and a data-access object with appropriate CRUD operations. In the standard line of the case study we use JDBC to prepare SQL statements and run them on a database, but students are encouraged to use other technologies (such as Hibernate) if they have experience with them. To show to the students that we can generate multiple output artefacts from the same model the case study requires that the students would also generate a database schema creation script for a specific database (e.g. Java Derby).

We chose templates as the basic generative approach, in particular the Velocity templating engine. Both entity and DAO classes are written as templates that are instantiated using the in-memory model obtained by parsing the DSL. However, we require the database schema script to be generated using the direct transformation approach. We want the students to realize that sometimes (for very simple output artefacts or when the static portion of the generated source code is relatively small) the template-based approach is unnecessary complex and it is more appropriate to use program transformation.

B. Constraints Language

The second iteration introduces another pragmatic solution – internal language based on a host GPL. We want to show to the students that if syntactic restrictions posed by the host GPL are not a problem, an internal DSL can significantly decrease parser implementation costs.

In this iteration the students define a new language that have to be composed with the entities language implemented in the previous iteration. Since the new language models an extension of the entities domain the two languages need to be composed. Thus the students are introduced to language composition on models. And finally, in the process of code generation we show them that templates can be composed, too. Template composition can be used to modularize and simplify the templates.

1) Abstract Syntax: The second iteration extends the problem domain with property constraints. In addition to property name and type, we want to be able to specify constraints about properties. For example, the value of a particular property might be required, it might have restrictions on range or length, etc. Students will use Java-based internal domain-specific language for constraints specification. Instead of extending the LineParser they will implement a new constraints language that will be composed with the entities language.

Since the constraints internal language is a new language, it has its own model. The model of the constraints language is shown in Figure 4. Classes highlighted in red represent references to the entities language. `EntityRef` and `PropertyRef` classes have both a `name` attribute that refers to an Entity concept and to a Property concept of the original entities language, respectively. For the purpose of the composition the entities language model has to be extended, too. The `Property` class representing the Property concept of the language gets a new attribute with a list of its constraints (similarly as the `PropertyRef` class).

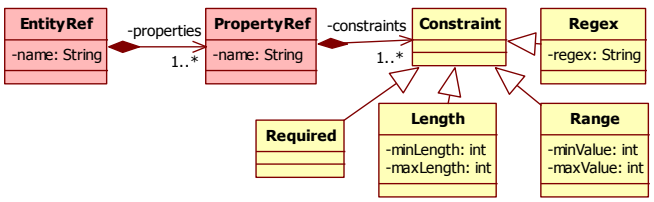

Fig. 4. The constraints language model

In addition to parsing, the constraints language have to be composed with the entities language. Students have to implement a method that validates references from the constraints language so that they refer to existing entities from the entities language model. After the validation they have to compose both models. That basically means that they have to assign constraints from the constraints language model to the appropriate properties objects from the entities language model.

2) Concrete Syntax: The second iteration of the teaching process is aimed at internal languages. Instead of imple-

menting their own parser, the students are shown how to reuse the compiler of the host general purpose language. This pragmatic solution is a façade to the language model that can be used to build constraints language expressions using domain-specific concrete syntax. In Listing 2 there is an example of a sentence specifying constraints on the name property of the Employee entity from Listing 1. The example specifies that every Employee must have a name and it cannot exceed 30 characters.

Listing 2. Constraints for the Employee entity in the constraints language

```
public class Constraints extends ConstraintBuilder {
  protected void define() {
    entity_ref("Employee",
      property_ref("name",
        required(),
        max_length(30)));
  }
}
```

The façade to the language model is called expression builder. Expression builder in the *CrudComp* case study is implemented as a Java class that provides creation methods with names from the problem domain. Part of the implementation for constraints expression builder is shown in Listing 3. It is implemented using the *nested methods* design pattern of internal languages inspired by Fowler [4].

Listing 3. Expression builder for the constraints language

```
public abstract class ConstraintBuilder {
  private List<EntityRef> entities = new ArrayList<EntityRef>();
  private Model model;

  protected abstract void define();

  protected void entity_ref
      (String name, PropertyRef... properties) {
    entities.add(new EntityRef(name, properties));
  }

  protected PropertyRef property_ref
      (String name, Constraint... constraints) {
    return new PropertyRef(name, constraints);
  }

  protected Required required() {
    return new Required();
  }
  :
}
```

3) Semantics: From the viewpoint of the constraints language semantics the students have to extend the DAO implementation to add a test method that validates objects of entity classes to match the specified constraints. Here we introduce another concept – template composition. For each constraint there should be a template just with the corresponding test. For example, in Listing 4 there is a Velocity template for the *required* constraint that tests whether an attribute of an entity class is not null (the `toUCIdent` method transforms the name into upper case identifier). In the test method of the DAO template there is a loop that goes through all the constraints assigned to the properties of the current entity class and instantiates and includes the appropriate template for each found constraint. This way we can avoid multiple 'if-else' conditional branches in the DAO template.

Listing 4. Test template for the required constraint

```
if(object.get${generator.toUCIdent($property.name)}() == null) {
  throw new ValidatorException("Property '$property.getName()'"
      + " of entity '$entity.getName()' is required.");
}
```

C. Entities Language with References

The third iteration moves the focus to the traditional MDSD tools. Now a new parser is not implemented in an ad hoc manner, but the students are supposed to work with a parser generator. The previously used approaches were supposed to show to the students that a simple DSL can be easily built without a lot of knowledge about the language theory. This iteration is used to show them that with modern approaches to parser generation, generating a parser is not difficult and for a non-trivial language it is much more effective than writing an own implementation.

To keep the course pragmatic we favoured model-based approach to parser generation (introduced in [5]). Students use the YAJCo [5] model-based parser generator that considers the object-oriented model of the entities language to be the specification of the language abstract syntax. Thus the students do not have to explicitly worry about the language grammar (although we show them the correspondence between the EBNF-based and model-based grammar specification). Using model-based parser generation does not necessarily require extensive knowledge of language and grammar theory.

1) Abstract Syntax: In the third step the problem domain is extended with relationships between entities. An entity uses a reference to other entity to express a relationship. For example, an employee works in a specific department. This relationship will be expressed by a reference from the Employee entity to the Department entity.

In this iteration we create a new parser for an external DSL that supports entities, constraints and references. The language model from previous iterations is reused thus mimicking agile evolution of the MDSD solution. Prototype parsers implemented in previous iterations are discarded, but the model and the generators are still used. A new parser populates the same `Model`, `Entity`, `Property`, and `Constraint` classes that were previously instantiated by the `LineParser` and the `ConstraintBuilder`. From the viewpoint of the language model, only the `Reference` class is an addition. Figure 5 shows the new `Reference` class with relations to the existing `Entity` class.

2) Concrete Syntax: The new entities language with references is an external language that is parsed by a generated parser. YAJCo parser generator uses the language model expressed by Java classes as a definition of the language abstract syntax. In addition to the model, students have to specify the concrete syntax of the language so that sentences like the one in Listing 5 can be processed.

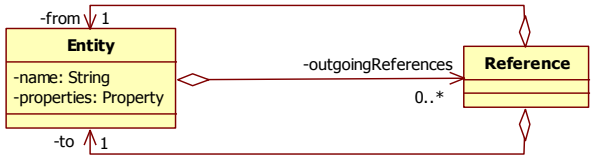

Fig. 5. `Reference` class and its relationship with the `Entity` class

Listing 5. Sentence in the entities language with references

```
entity Department {
  name : string required, length 5 30
  code : string required, length 1 4
}
entity Employee {
  name : string required, length 2 30
  age : integer
}
reference from Employee to Department
```

YAJCo uses Java annotations to associate concrete syntax patterns with the language abstract syntax expressed by Java classes. Each constructor of a language model class is considered an alternative in the grammar rule for expanding the language concept represented by that class. A constructor is a mean for creating an object from formally defined input. Annotations are used to associate concrete syntax to these rules, e.g., keywords with `@Before`/`@After`, number of occurrences with `@Range`, etc. Listing 6 illustrates modelling an expansion rule for the Entity concept of the class by annotating parameters of the constructor (for illustration of the duality there is an EBNF-based version of the rule).

Listing 6. Entity concept concrete syntax expressed by YAJCo annotations

```
public class Entity implements Named {
  :
  // Entity –> 'entity' NAME '{' Property+ '}'
  public Entity(@Before("entity") String name,
                @Before("{") @After("}") @Range(minOccurs=1)
                Property[] properties) {
    this.name = name;
    this.properties = properties;
  }
  :
}
```

In this point the students already have the abstract syntax of the language – they have the language model. To generate a parser they only need to properly annotate constructors of the model classes to specify the concrete syntax of the language.

3) Semantics: This iteration introduces only small additions to the generated artefacts. The students have to alter the database schema script generator, entity class template and DAO template to support references between entities.

D. User Interface Specification Language

The last iteration we use to teach the concept of generic languages (called Commercial-Off-The-Shelf by Kosar et al. in [6]). If a language designer keeps the syntactic restrictions defined by a generic language he/she can then reuse its generic parser. Generic languages (XML, YAML, .properties, etc.) are currently very popular in industry, especially for configuration languages [7]. This popularity is the reason why we believe that generic languages should be a part of an MDSD course. From the viewpoint of code generation and templates we use this iteration to show how the templates can be reused.

1) Abstract Syntax: In the last iteration the problem domain is extended to consider the user interface of the CRUD application. Entity objects are presented to application users in tables, each of which has a set of columns corresponding to entity properties. Not all the properties of a particular entity have to be presented and therefore there does not necessarily have to be a column for each property. To support creating and editing entity instances, a form has to be specified. Again, for each property, a field in the form can be defined.

The language model for the UI specification DSL includes new classes that describe concepts of a CRUD user interface. In Figure 6 there is a class diagram showing the language model of the UI specification language. A user interface consists of tables and forms for the entities. Tables and forms are special cases of a dialog. Each dialog has its components; in case of tables those are columns and in case of forms the components are fields. The `Dialog` and the `Component` classes have attributes prepared for composition with the entities language.

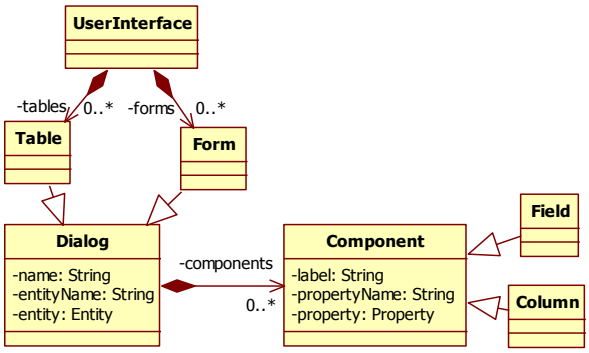

Fig. 6. User interface specification language model

2) Concrete Syntax: The concrete syntax of the UI specification language is XML-based. For parsing the language we chose the Java Architecture for XML Binding (JAXB). JAXB enables to marshal (serialize) a Java object tree into a corresponding XML document and to unmarshal (deserialize) an XML document into its in-memory object oriented representation. The classes that JAXB works with are indeed the model of the XML-based language that specifies its abstract syntax. JAXB uses *convention over configuration* design pattern to assume a concrete XML-based syntax of the language. For example, by default, JAXB maps class attributes to XML elements with the same name (mapping options between XML format and object trees are discussed in detail in [8]). The JAXB schemagen tool that can be used to generate XML Schema Definition for the XML language.

Listing 7 shows a simple user interface specification for the CRUD application with Employee entities using an XML-

based notation. Again the students' task is to annotate the language model (Figure 6) so that the JAXB would be able to marshal and unmarshal instances of the model to XML documents with the format shown in Listing 7.

Listing 7. XML-based user interface specification language

```
<ui>
  <form name="EmployeeForm" entity="Employee"
        label="Employee">
    <field property="name"/>
    <field property="age"/>
  </form>
  <table name="EmployeeTable" entity="Employee"
        label="Employee" editFormDialog="EmployeeForm">
    <column property="name"/>
    <column property="age"/>
  </table>
</ui>
```

JAXB annotations are used to specify deviations from the default mapping. An excerpt from the `Dialog` class with JAXB annotations in Listing 8 represents an example of a mapping definition. `@XMLTransient` annotations exclude program elements from mapping. E.g. the Dialog class itself is excluded since its descendants will suffice. `@XmlID` annotation specifies that the `name` attribute is an identifier of `Dialog` (or its subclasses) objects. `@XmlAttribute` overrides mapping to XML element and specifies that the `name` attribute of the `Dialog` class will be mapped rather to XML attribute.

Listing 8. Dialog class annotated with JAXB annotations

```
@XmlTransient
public abstract class Dialog implements Named {
  @XmlID @XmlAttribute(required=true)
  private String name;
  @XmlAttribute(name="entity", required=true)
  private String entityName;
  @XmlTransient
  private Entity entity;
  @XmlAttribute(required=true)
  private String label;
  @XmlTransient
  private Component[] components;
  :
}
```

3) Semantics: The last iteration finishes the CRUD application generator. The students will implement generators that will provide a user interface for the data tier generated by the generator implemented in the previous iterations. Currently as the standard line in the case study we use a console-based user interface so that the project would be simpler and at least partially manageable even for under-average students. The standard console-based UI solution requires providing templates for forms and tables. In addition, the students have to write a simple template for the main class of the application that provides the main menu for using it. Of course, we encourage the students to rewrite the project to support other types of user interfaces – we have seen multiple web-based UIs (e.g., HTML+JavaScript, Java Server Faces), desktop UIs based on Swing and also mobile clients (e.g., Windows Phone 8 communicating with server through web services, Blackberry) developed by our students.

From the viewpoint of generation techniques we ask the students to reuse the templates. The UI has to validate users input to avoid violating constraints on entity properties[2]. Here they have to reuse the constraints validation templates written in the second iteration (e.g. the template shown in Listing 4). This way they can see that a good decomposition of templates can also support template reusability.

V. Evaluation

To determine the impact of using MDSD in our course, we administered a survey to the students in our classes. 58 students responded the survey. Following questions were used in the questionnaire

A Single choice questions:

1. What were your experiences with model-driven software development (MDSD) before this course?
   (a) I have not heard of it before, (b) I have heard of it before but I have never used it, (c) I have already used this approach before this course.
2. Do you think you understood MDSD?
   1 - Strongly agree, 2 - Agree, 3 - Disagree, 4 - Strongly disagree.
3. Would you use the techniques learned in this course in practice?
   1 - Strongly agree, ..., 4 - Strongly disagree.
4. Were you satisfied with the iterative way of development used in the course?
   1 - Strongly agree, ..., 4 - Strongly disagree.
5. Rate the amount of work needed to complete the project solved in the course.
   (a) Significantly more than in other courses, (b) More than in other courses, (c) Less than in other courses, (d) Significantly less than in other courses.
6. The course belongs to your:
   (a) favourite subjects, (b) rather favourite subjects, (c) rather not favourite subjects, (d) not favourite subjects.

B Open text questions:

7. What did you like about the course?
8. What is the biggest problem you had during the course?
9. What would you change about the course?
10. Which of the learned techniques would you use and in what situations/projects/platforms?

As the reader can see, the first two questions of the questionnaire are oriented to students' knowledge about MDSD before and after the course. The 2nd and 10th question are targeted to practical usage of the learned knowledge. The rest

[2]In this simple console-based application the validation both in data tier and in UI is redundant, but we want the students to have an opportunity to reuse the templates written for constraints. In the real world, most of the common CRUD applications are web based. In web input, validation in UI forms is important for user experience. And duplicate validation on server is necessary if the server exports services that can be used to create or update entities.

of the survey addressed the course, it's form and the problems that the students might have had during the course.

Results

The results we obtained from the first two questions revealed that most of the students (57%) have never heard of MDSD and only 5% have used an MDSD approach before the course. After finishing the course, almost 86% of the students think they understand MDSD and only one student feels s/he does not understand MDSD at all.

More than a half of the students (almost 58%) think that they will use the MDSD techniques in practice. Here we have to note, that not all of our students are programmers and many students are focused on computer networks for example. According to the answers of the 10th question, more than a half of all students (51%) specified also relevant examples of using specific techniques in practice. This fact implies that *more than a half of the students sufficiently understood MDSD principles and techniques, they can distinguish between them and know how to use them in practice*.

The results of the 4th question shows that majority of the students (93%) liked the iterative approach used in our course.

It was surprising and gratifying for us to learn that although 88% of students thought the course puts an excessive amount of work on them, however 70% marked the subject as their favourite or rather favourite.

The problems that our students encountered most frequently were mainly misunderstanding of several tasks in the course materials (30%), technological issues (IDE, operating system compatibility, etc.) or Java (25%). *17% of students had no problem with the course*. Only a little number of students had problem with the techniques used - YAJCo (3 students), annotations (2 students) or velocity (7 students).

Although the students had issues, the results of the 9th question show that *more than a half* of them (52%) felt that they *would not change the course materials at all*.

The results obtained in the open text section showed that the students liked the iterative approach very much and they are satisfied with the consultation during exercises. Many of them marked the exercises as useful and they liked the implementation. Some of them like the various techniques used and favour the possibility of using the learned techniques in practice.

We can conclude that the course is successful and orientation to the practice had motivating impact on the students. Problems lied mainly in the formulation of several tasks, which were hard to interpret for weaker programmers. For this reason, we introduced discussions and evaluations of concrete tasks into our course materials, to be able to obtain task-specific feedback and improve the course materials in the future.

VI. Related Work

The motivation for teaching MDSD at our university is based on its promises of narrowing the semantic gap between problem and solution domains. Selic [9] argues that these benefits of using models are even greater in software than in other engineering disciplines (due to less diversity in skills needed for the complete MDSD implementation). Introduction of the MDSD course at our university was a response to the studies and works that on the one hand proclaim the benefits of MDSD, but on the other hand state that MDSD is given little attention in education (e.g., an early work by Cowling [10]). The main problem with MDSD teaching at our university is that myths discussed in Section I were and still are strongly rooted among our students. Although there are numerous articles describing research challenges in MDSD (e.g., work by France et al. [11]) we faced the problem of MDSD unpopularity among the students. And our students considered most of the scientific papers on the topic as just proofs of those myths (they usually deal with the highly specific problems). In our teaching approach we tried to extract the fundamental MDSD principles and show them to the students on simple pragmatic examples. The principles had to be directly applicable in practice (considering the small scale application, application in the agile methodologies, etc.).

Considering taught principles we explain the MDSD topic from the viewpoint of the language-oriented programming (see Section II). Although this viewpoint covers basically the same challenges and benefits as rather "classic" MDSD teaching approaches such as the one by Clarke et al. [12], our approach is more language-centric. We decided to extensively cover also the topic of formal languages, since currently in the industry there is ubiquitous need for developing and working with little languages (especially configuration languages [7]). Not only this course teaches the students useful knowledge, but it also serves as a motivational factor; the most important attribute of a course for students is whether they will be able to apply the learned topics in their future career.

Problem with used tools in MDSD teaching was discussed in multiple scientific works. There are cases in which teachers chose complex MDSD tools and they do not report any significant problems with students using complex solutions. For example, Tekinerdogan [13] and Clarke et al. [12] used Eclipse Modeling Project (EMP) tools, Pareto [14] used Microsoft DSL toolkit. However, there are reports of students having problems with working with such complex tools. For example, Batory et al. [15] tried to use the Eclipse Modeling Framework (EMF), but their students were overwhelmed by the technology. The failure to successfully understand and work with the EMF resulted in, using words of Batory et al., "a bitter taste" for them, and worse, even their students. We did not use a single complex tool to defy the myths about the solely large scale MDSD application and the need of massive tool support.

Schmidt et al. [16] identified three approaches to MDSD teaching:

- *Purely theoretical approach* that focuses on theoretical knowledge and neglects the practical exercise of the MDSD principles by students themselves.
- *Tool-supported approach* is a teaching approach that uses a single complex MDSD tool (e.g., EMF in [12]).

- *Practical approach* that focuses on underlying concepts rather than the use of a concrete tool.

Schmidt et al. use the practical approach in which they ask students to implement the generator tool themselves. They favoured this approach to the tool-supported approach since with the practical approach students have to directly apply the MDSD elementary principles themselves. Using a complex MDSD framework risks that some of the basic principles might be encapsulated by the framework and thus hidden from the students. Although this motivation differs from ours (we did not use a complex tool to show that MDSD can be applied without a massive tooling support) we ended with the very similar approach focused rather on principles than on tools.

Considering the domain of the course project, the used domain usually differs from work to work. For example, Mosterman [17] uses the domain of embedded systems, Clarke et al. [12] use the domain of communication services, or Batory et al. [15] let the students choose a domain of their interest. For the case study in our course we selected the domain of CRUD applications for two reasons, (1) it is a well-known domain for our students, and (2) it is widespread in practice (considering frameworks such as Spring Roo or Ruby on Rails). While the usage of a well-known domain does not bother the students with unnecessary learning load, the fact that the domain is widespread in the industry serves as a motivational factor.

From the viewpoint of the teaching approach, most of the articles report using classic development with a single iteration (e.g., Clarke et al. [12]). We use iterative approach to show the options in using MDSD for incremental, agile development and also to reduce the focus on a complex MDSD tool and to rather move it to MDSD principles. Iterative teaching approach is also used by Schmidt et al. [16]. They use the iterative approach for the same reason as we do; they want to focus on MDSD principles rather than on tools. In the first iteration their students implement their own generator tool, in the second iteration they extend the tool, and only in the last iteration they implement a language using a complex MDSD tool.

VII. Conclusion

In this paper we have presented our approach to teaching model-driven software development. The goal of our course is to explain the basic principles and concepts of model-driven and generative development. These concepts are illustrated using several different practical tools and techniques that can be used in different combinations and in projects of different scale. The presented approach could be also inspirative when adapting the model-driven approach in a software project. Thanks to the case study students can acquire practical experience with each of presented techniques during the course. Iterative character of development also provides insight into the use of MDSD as a part of the development process.

Acknowledgment

This work was supported by project KEGA No. 019TUKE-4/2014 Integration of the Basic Theories of Software Engineering into Courses for Informatics Master Study Programmes at Technical Universities - Proposal and Implementation.

References

[1] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700

[2] A. Demir, "Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development," in *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, ser. MBD-MOMPES '06. Washington, DC, USA: IEEE Computer Society, 2006. doi: 10.1109/MBD-MOMPES.2006.5. ISBN 0-7695-2538-5 pp. 75–83.

[3] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-61622-X

[4] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010. ISBN 0-321-71294-3

[5] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation Based Parser Generator," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 291–307, Apr. 2010. doi: 10.2298/CSIS1002291P. [Online]. Available: http://www.comsis.org/archive.php?show=ppr230-0911

[6] T. Kosar, P. E. Martínez López, P. A. Barrientos, and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Inf. Softw. Technol.*, vol. 50, no. 5, pp. 390–405, Apr. 2008. doi: 10.1016/j.infsof.2007.04.002

[7] M. Nosáľ and J. Porubän, "XML to Annotations Mapping Patterns," in *2nd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), J. P. Leal, R. Rocha, and A. Simões, Eds., vol. 29, 2013. doi: 10.4230/OASIcs.SLATE.2013.97. ISBN 978-3-939897-52-1. ISSN 2190-6807 pp. 97–113.

[8] R. Lämmel and E. Meijer, "Revealing the X/O impedance mismatch: changing lead into gold," in *Proceedings of the 2006 international conference on Datatype-generic programming*, ser. SSDGP'06. Berlin, Heidelberg: Springer-Verlag, 2007. doi: 10.1007/978-3-540-76786-2_6. ISBN 3-540-76785-1, 978-3-540-76785-5 pp. 285–367.

[9] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003. doi: 10.1109/MS.2003.1231146

[10] A. J. Cowling, "Modelling: a neglected feature in the software engineering curriculum," in *Proceedings of the 16th Conference on Software Engineering Education and Training 2003*, ser. CSEE T 2003, March 2003. doi: 10.1109/CSEE.2003.1191378. ISSN 1093-0175 pp. 206–215.

[11] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007. doi: 10.1109/FOSE.2007.14. ISBN 0-7695-2829-5 pp. 37–54.

[12] P. J. Clarke, Y. Wu, A. A. Allen, and T. M. King, "Experiences of Teaching Model-Driven Engineering in a Software Design Course," in *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'09. Denver, Coloardo, USA: IEEE Computer Society, 2009.

[13] B. Tekinerdogan, "Experiences in teaching a graduate course on model-driven software development," *Computer Science Education*, vol. 21, no. 4, pp. 363–387, 2011. doi: 10.1080/08993408.2011.630129

[14] L. Pareto, "Teaching Domain Specific Modeling," *Symposium at MODELS 2007*, p. 7, 2007.

[15] D. S. Batory, E. Latimer, and M. Azanza, "Teaching Model Driven Engineering from a Relational Database Perspective," in *MoDELS*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, Eds., vol. 8107. Springer, 2013. doi: 10.1007/978-3-642-41533-3_8. ISBN 978-3-642-41532-6 pp. 121–137.

[16] A. Schmidt, D. Kimmig, K. Bittner, and M. Dickerhof, "Teaching Model-Driven Software Development: Revealing the "Great Miracle" of Code Generation to Students," in *Sixteenth Australasian Computing Education Conference (ACE2014)*, ser. CRPIT, J. Whalley and D. D'Souza, Eds., vol. 148. Auckland, New Zealand: ACS, 2014, pp. 97–104. [Online]. Available: http://crpit.com/confpapers/CRPITV148Schmidt.pdf

[17] P. Mosterman, "Automatic Code Generation: Facilitating New Teaching Opportunities in Engineering Education," in *36th Annual Frontiers in Education Conference*, Oct 2006. doi: 10.1109/FIE.2006.322699. ISSN 0190-5848 pp. 1–6.