Computer Science and Information Systems 12(2):683–705 DOI: 10.2298/CSIS140107022P

Teaching Pragmatic Model-Driven Software Development

Jaroslav Porubän, Michaela Bačíková, Sergej Chodarev and Milan Nosáľ

Technical University of Košice, Department of Computers and Informatics
Letná 9, Košice, Slovak Republic
{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. For long, our students were victims of these myths and considered MDSD impractical and only applied in academy. In this paper we discuss these myths and present our experience with devising an MDSD course that challenges them and motivates students to understand MDSD principles. The main contribution of this work is a set of MDSD teaching guidelines that can make the course pragmatic in the eyes of students – programmers. These guidelines introduce MDSD from the viewpoint of a programmer as a pragmatic tool for solving concrete problems in the development process. In our MDSD course we implemented the presented guidelines. The course shows several techniques and principles of model-driven development in multiple incremental iterations instead of concentrating on a single tool. At the same time we unite these techniques by using a dynamic visualisation tool that shows to the students the whole infrastructure in the big picture. The course is implemented as an iterative incremental MDSD case study. The paper concludes with a survey performed with our students that indicates positive results of the approach.

Keywords: Model-driven Software Development, Teaching, Case Study, Iterative MDSD, Visualisation Tool, Pragmatic MDSD, Domain-Specific Languages.

1. Introduction

Model-driven software development approach (MDSD) promises increase of development speed and quality of resulting software by the use of formal model as a basis for system's implementation [38]. Understanding MDSD by our students, however, suffers from several myths that hamper its adoption. In this paper we present these myths about MDSD as the motivational context to our work.

The core of our work is the set of guidelines that we devised to keep an MDSD course pragmatic to challenge presented myths. These guidelines emphasize iterative incremental MDSD process and employment of common industrial tools and techniques integrated together instead of full-fledged MDSD workbenches. In context of the MDSD theory, we focus mainly on two topics: domain-specific languages (DSL) and code generation. DSLs deal with processing of the model input to an in-memory representation. Code generation uses the in-memory model to emit target code. We use this minimal MDSD skeleton to show the students that they can get MDSD benefits even without the complexity of MDSD "big guns". Other topics, such as model-driven architecture, software factories, meta-modelling, MDSD tools (EMF, MPS, etc.), and others, are covered only briefly on lectures.

The contributions of this work are as follows:

- We devised a set of *MDSD course guidelines* that challenge common myths about MDSD and make the course more appealing to common programmers (Section 3).
- We explain our implementation of the guidelines that can serve as an example (Section 4). It can provide inspiration for others who want to design an MDSD course and are concerned with the fact that students are influenced by the myths stated below. We also present a short survey performed with our students to evaluate the potential of our approach (Section 6).
- We show how visualisation may help students to understand the whole picture of the MDSD even when the course employs iterative approach with multiple tools and techniques integrated into a full MDSD solution (Section 5).

2. Myths about MDSD

We identified several myths that hamper the adoption of the MDSD approach. Our students usually come to our MDSD course with these myths deeply rooted in their minds.

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 [8]). 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 [21]. As latest research shows, most of the successful MDSD projects were developed this way [16]. 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 introducing MDSD may not require significant changes in the architecture of the system as a whole.

Myth 2: MDSD requires massive tool support

MDSD is often associated with integrated modelling tools or language workbenches. These tools cover development of a meta-model, a domain-specific language used to express models and a generator that produces runnable code based on a model. modelling tools can 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 [1]. 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 [43], etc.). This approach allows looser coupling and greater flexibility in the choice of tools.

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 and this approach is usually more successful in industrial practice [16]. 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 the previous paragraphs greatly simplifies such iterative development process and allows using MDSD along with common agile methodologies.

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 [15]. Most of the examples, however, represent small-scale MDSD applications which include:

- Generators of database schemas (e.g., DOMMLite [7], or database refactoring tools [24]) 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 (e.g., for business applications [30]).
- IDE plugins for specific technologies that are able to generate skeletons of repetitive artefacts (e.g., GWT plugin for IntelliJ Idea that generates 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, MDSD application is often hidden from a programmer by libraries and frameworks that allow to specify behaviour using a model without knowing any 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.

3. Pragmatic Model-Driven Programming

In our course we wanted to challenge the aforementioned MDSD myths to overcome students' scepticism. 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 and in correspondence with the use of MDSD techniques in industrial practice as described by Whittle and Hutchinson [42].

This is why we are referring to *pragmatic* model-driven programming. We have come to the following guidelines for the case study for students that allow a course designer to fulfill the stated goals.

Use cooperating interchangeable tools and techniques instead of a single complex tool. Students should understand the basic MDSD principles as they have a much higher level of applicability than any concrete tool. Therefore the course has to emphasize principles above any concrete tools. For this reason we have decided not to use traditional MDSD tools (e.g. based on MDA [17] or Eclipse Modelling Framework) in our case study and develop the project ground-up instead. Although we demonstrate MDSD development of a complete system, parts of the system are modelled and generated separately showing different scales of modelling and using multiple tools and techniques. Students should also see different combinations of these techniques and understand their advantages and disadvantages.

Divide the complete solution into small partial solutions. Each lesson can focus on one such solution, for example parsing or code generation using templates. This provides more in-depth understanding of each part of the MDSD infrastructure thus emphasizing principles over tools.

Compose several small languages instead of building a single large language. This shows that it is possible to use MDSD principles even for a small aspect of a system and then to combine it with hand-written code or other languages (more on interoperability of multiple DSLs in a single solution can be found in [28]). Language composition also allows to keep domain-specific language really focused on their domains. This approach also corresponds to how MDSD is used in practice [16].

Put the separate solutions back into the whole story. To overcome fragmentation that can be caused by the previous guidelines, it is important to provide also a high level overview of the whole solution and roles played by its parts. Also, the usage of each tool/technique has to be backed up by a presentation of the problem context and reasoning for the choice of the given tool/technique. The best solution is an interactive visualisation of the language processing pipeline where each process and artefact is directly interconnected with students code.

Show the iterative incremental approach. This not only shows the integration of MDSD into agile processes, but also allows to teach different techniques on the same case study. Iterative approach can also helps understanding the basic MDSD principles as they are repeated in each iteration. Incremental solution helps to understand that the MDSD can be applied in a small scale as well.

Use technologies that students already know as the basis. It is difficult for students to understand new methodology if they are overwhelmed by the details of new tools they need to use (accidental complexity). The best is to choose programming language that the students know from previous courses along with commonly used tools.

Focus on common industrial techniques and technologies. We should illustrate common MDSD principles using realistic examples, tools and approaches that are and can be used in practice (even in small scale). 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 (along with tools) should be applicable in a wide range of situations. These are tools that students would probably use in their career even if they would not build a complete MDSD solution. Learning tools that are used in industry and implementing realistic solutions also increases students' motivation.

Explain how to switch to MDSD (and back). Using common industrial techniques and tools shows the students that they do not have to learn new techniques and tools to adopt MDSD; all that is important are the principles. Low barrier of exit is also important as it decreases the risk of being locked in the solution that can turn unsuccessful. The use of general-purpose language to define the model greatly helps to solve problems of low entry and exit barriers.

Use the model-first approach. It should be shown that the model is the central and the most stable part of the solution. Explicitly defined model allows to develop parts of the solution separately and even replace them without changing the rest of the system. In addition, definition of explicit domain model is considered a good practice even outside of MDSD [11].

Show reuse. Reuse of artefacts is one of the fundamental software engineering principles. This principle should be practically demonstrated during the course by showing how proper decoupling allows to reuse parts of the implementation during the system development. Reuse helps the students to understand that MDSD manifests common engineering characteristics and it is easily applicable in practice.

Select a well-known and interesting domain. We cannot expect students to have experience with domains that are not widespread or directly related to their previous courses. Also, a good choice of a domain for the project can help students' motivation.

Always show industrial examples of MDSD adoption. There are many MDSD solutions in practice and many of the students have already used them without realising that they are working with an MDSD solution. We have to emphasize these examples to get students' attention and to persuade them that MDSD is not just an academic approach.

Build an executable system. The case study that students work on during the course should result in a complete executable system. This does not necessarily have to mean that the whole system has to be generated, however, an executable result will give the students more satisfaction with their work. The aim of this guideline is to motivate them.

We believe that the aforementioned guidelines challenge the myths listed in Section 1. By using cooperating interchangeable tools and techniques we show that MDSD can be implemented without using a single large MDSD tool such as JetBrains MPS. The same myth is challenged by using multiple common tools and technologies. The problem of students looking at MDSD as a solely large scale approach is challenged by dividing the MDSD solution into small partial solutions, composing several small languages and by a

practical explanation of switching to MDSD and back (using incremental development). The myth about MDSD not supporting agile development process is challenged by using iterative incremental approach, and again by explaining how to switch to MDSD and back. Finally, the myth about MDSD not being adopted in industry can be easily rejected by showing the students many industrial examples. Here a well-known domain for the case study can help, too. If the industry uses MDSD for that domain, the students can more easily accept the fact about industrial MDSD adoption.

Some of the guidelines were devised before we started our course, but many of them came as an experience only after first years of teaching MDSD. As an example, after interviewing the students we learned that even after they worked on a carefully selected domain, they still had problems seeing MDSD applied in practice. We realised that there is a myth about MDSD industrial adoption and since then we are always showing examples from practice (e.g., visual GUI editors in IDEs). Another experience learned later was the problem of solution fragmentation caused by division of solution into partial solutions: in the process of implementing the case study, many students lost the big picture and thus failed to fully grasp the basic MDSD principles.

In the next sections we present our course in more details. We use a single case study project that is developed in an iterative manner. We apply one of the standard approaches to MDSD that uses an explicitly defined meta-model [13]. Meta-model of the generated system used in the case study is gradually extended showing evolution and composition [25] of meta-models and languages during the MDSD development.

4. Course in Practice

Section 3 summarizes our experience with teaching MDSD to pragmatic programmers in form of guidelines or goals that have to be achieved. However, these guidelines are quite abstract. For a better understanding of our experience we use this section to explain details about our own MDSD course that can inspire other educators.

4.1. Methodology

The MDSD course at our university is taught to the first year's graduate students. They will most likely work as developers after graduation and therefore we wanted to focus the course on the pragmatic aspects of MDSD. From previous courses they all have experience with Java programming language and object-oriented programming, thus the Java language was an obvious choice as the case study platform to prevent accidental complexity. Both the language of the generator and of the generated applications is Java.

As a teaching methodology we use task-driven case studies (abbreviated: TDCS). TDCS is a methodology developed at our university and we already use it for several years on multiple subjects with very positive feedback from both the students and teachers [34]. Basically, TDCS combine task-driven teaching (focusing on student's task) with case studies (to provide a meaningful context to those tasks). An important part of the methodology is to prepare study guides that contain case study context and reasoning and content structured in objectives and tasks. Study guides are provided on-line to students, so they can work on their projects from home, too. During the course, each student works

individually on his/her own solution at home or in the class. More details about the TDCS methodology can be found in [34].

The course is taught in the second semester and it lasts 13 lessons in 1 semester, each of 2 hours per week. Students' progress is controlled in the class on a weekly basis to prevent procrastination and to help them with issues that raise during the development of the case study. We use the first lesson in the course to introduce the basic topics of the case study, to explain the motivation and context and to explain the course organisation[1]. To promote an *iterative incremental approach* we divided the case study into four relatively self-contained partial solutions. Each of these iterations lasts three lessons to finish. The first lesson is always dedicated to language processing and model manipulation (evolution, composition); the second lesson is dedicated to model-to-source-code transformations (code generation); and the third lesson is used to evaluation of students' solutions. Each iteration is graded by points and students are encouraged to add their own features for additional points to increase their motivation to go deeper into the problem. Since we keep each iteration relatively simple, the students have a working executable solution after each lesson. E.g., in the first lesson of each iteration, they implement a working parser and after the second lesson they implement a generator that works with a model generated by the parser. After each iteration the solution can be used to generate executable code artefacts.

4.2. Case Study

In the case study domain selection we were inspired by successful web development frameworks that employ the principle of *Convention over Configuration* to provide fast bootstrap for CRUD (create, read, update, delete) applications. E.g., in GRails framework it is sufficient to implement a data model and the framework provides the developer with a working CRUD application. CRUD applications are common and therefore known by students. The fact that there are solutions such as GRails or Spring Roo also helps us to show that MDSD is for this domain used in the industry as well. In their projects the students are expected to implement a generator for simple CRUD applications in Java. A concrete CRUD application is defined by a simple model of the domain in terms of entities, their properties and relationships.

We designed the architecture of the case study CRUD applications to be a 3-tier architecture. The CRUD applications have three architectural layers: user interface, service and data access. The architecture skeleton is depicted in Figure 1. These CRUD applications are simple Java console applications. The data layer allows to write entities into a relational database (we use Java Derby). Service layer is defined by simple data access object (DAO) interfaces and their implementations. And finally, the user interface is implemented as a simple console interface. The implementation is kept simple so that every average student would be able to finish it.

The case study is divided into 4 relatively self-contained parts. Implementation of a DSL for each of these parts is a goal of one iteration in the course. Using Java and common

[1] An English version of the study guides can be found at `http://hornad.fei.tuke.sk/~bacikova/MaGSA/01/`. Although we currently use a newer updated version of the study guides, these should serve as a sufficient illustration (or inspiration, may the reader be considering to adopt our approach).

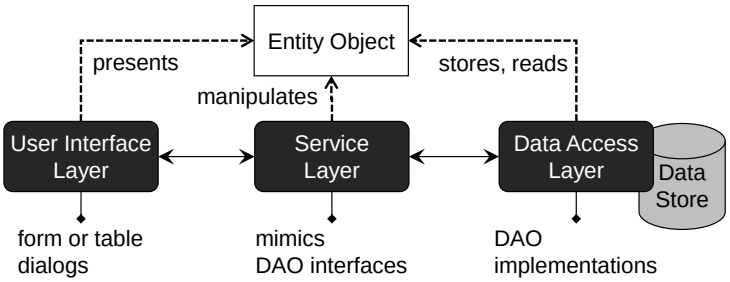

Fig. 1. Multi-tier architecture used by *CrudComp* CRUD applications

tools shows that MDSD can be adopted relatively easy. Using iterations to implement the case study incrementally shows that a developer can generate only a part of the system and the rest can be implemented by hand, thus promoting easy switch back from the MDSD and one can even combine multiple approaches in the context of the same system. Incremental approach also enables agile development.

The CRUD applications generation was divided into the following four parts that correspond to the aforementioned iterations:

1. *Simple data layer generation.* The generated artefacts are classes for data entities (e.g., a `Patient` class in case of CRUD application for the domain), simple DAO interfaces, DAO implementations and an SQL database table creation script. The solution of this iteration cannot generate a whole system, but it can generate a bootstrap data tier for a CRUD application.
2. *Entity constraints support*. Entity objects that are handled by CRUD applications should be validated to conform to domain rules. E.g., a doctor might be required to have at least five years of experience in his/her field. The iteration enriches DAO implementations to validate the entity objects before persisting them to the database.
3. *References between entities.* The third iteration again adds generated code to the data and business layers. The first two iterations have supported simple entities. This one allows relationships between them by introducing references. A doctor can have associated patient records, etc.
4. And finally, the fourth iteration introduces a console-based *user interface*. This includes several new source code artefacts. First, a menu for interacting with the application showing options for each entity handled by the application is created. Then, for each entity there is a table that presents all the instances and a form for editing and creating instances.

The whole case study also includes a framework that implements universal code parts shared by all generated CRUD applications. The framework provides a base class for DAO implementations, base classes for forms and tables and code for database connection. Students are also provided with skeletons and parts of the partial solutions they have to work on. For example, when generating a DAO implementation students are provided with a template skeleton and they are supposed to finish just a few small parts of the template. By seeing a part of the solution they can get to the problem and new technologies

more easily. However, later in the course they are given harder tasks. E.g., in the end, when working on the console UI they have to write the whole templates for forms and tables. Although they are provided with parts of the solution the whole case study is still simple enough to comprehend the whole picture.

4.3. Tools and Approaches

All four case study parts revolve around the *model* of CRUD application. Model defines how should the entities in the application look like, what are their properties, types of the properties, relations between entities, etc. The definition of a concrete model expressed by a meta-model. For the meta-model definition we use Java classes, again, because we do not want to bother the students with overwhelmingly complicated model definition in a completely new and previously unused language or technology. Therefore each concrete model consists of in-memory Java objects. These objects are created by parsers of DSLs implemented during the course.

While each iteration uses a different technique and tools, they all share the same tooling infrastructure used in MDSD (see Figure 2). 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. Meta-model in Java enables us to use multiple approaches and tools interchangeably; we utilize this fact between iterations.

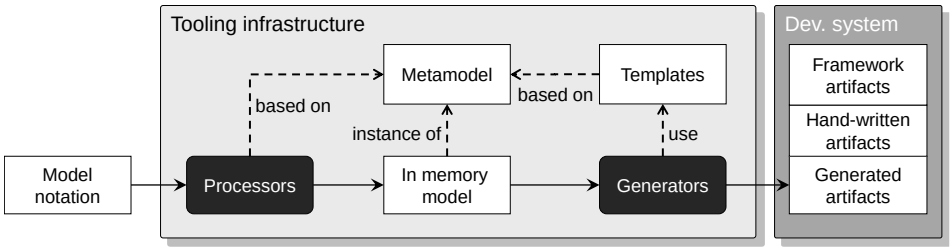

Fig. 2. Tooling infrastructure of our MDSD case study solution

In this section we discuss the concrete tools and approaches taught in the course. Each subsection is devoted to one of the course iterations. However, for more details about the case study project we refer the reader to [32].

Data Layer Generation For the data layer generation students need a model that describes 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, and a type. For the purpose of expressing these information the students have to implement an external DSL that we call the *Entities DSL*.

The Entities DSL is implemented as 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 or language workbenches

Listing 1.1. Two entities specified in a file-based Entities language

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

could just complicate the matter (learning curve, etc.). To make the implementation of the entities language as simple as possible we exploit the file system for the concrete syntax: description of each entity is stored in a separate file and its name denotes the name of the entity. 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. An example of a language sentence is shown in Listing 1.1 (angle brackets denote files and directories).

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* (widely used Velocity template engine) and *direct transformation* (pure Java) approaches described by Fowler [11]. Again, we want the students to understand when a generation using direct transformation is enough and when we can simplify generation with templates.

In this 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 1.1 they are supposed to generate a Java entity class and a data-access object with appropriate CRUD operations by applying the template-based generation approach. 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. This way we are nurturing individual approach without adding accidental complexity. 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 by applying the direct transformation approach.

Combination of the delimiter-directed parser and direct transformation shows to the students that they do not necessarily need to learn any new language or tool to get started with MDSD. The most important are the MDSD principles. Introduction of templates and later other specialized tools helps them understand that these tools are nothing to be afraid of and that generation can be much more effective with those tools.

Entity Constraints Support 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 on properties. For example, the value of a particular property might be required, it might have restrictions on range or length, etc. To support constraints we do not simply extend the existing Entities DSL; we rather introduce a new DSL specialized for constraints – the *Constraints DSL*. The languages are composed on level of models, so a single complete model is created as the result. This way the students can see *language composition on models* in practice and we can also introduce new MDSD techniques that can be used by the students. The Constraints DSL is an *internal language* based on Java. We want to show to the students that if syntactic restrictions posed by the host general purpose language (GPL) are not a problem, an internal DSL can significantly decrease parser implementation costs. Constraints DSL is a façade to the language model that can be used to build constraints language expressions using domain-specific concrete syntax. In Listing 1.2 there is an example of a sentence specifying constraints on the name property of the Employee entity from Listing 1.1. The example specifies that every Employee must have a name and it cannot exceed 30 characters.

Listing 1.2. Constraints for the Employee entity in the Constraints DSL

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

Assembling the complete model from two different notations helps the students to understand that in MDSD there may be multiple DSLs combined together to achieve a single goal. Many times there already is an existing DSL that is perfect for a given domain, but the users need to do some things that are beyond its domain. Instead of polluting the existing DSL, it can prove efficient and useful to just implement a new DSL for the new domain, and then compose both DSLs together. This way the 'good old' DSL does not have to become a GPL to be more usable.

And finally, in the process of code generation we show the students that templates can be composed, too. *Template composition* can be used to modularize and simplify the templates. For each constraint the students have to define a template with appropriate test. The template for the DAO implementation implements a `test()` method by including appropriate templates according to the constraints model.

References between Entities The third iteration moves the focus to the traditional MDSD tools; we introduce a parser generator. The delimiter-directed parser and expression builder for the internal DSL created in the previous iterations are substituted with a generated parser. 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 a custom implementation. In this iteration a completely new language is designed – the *Entities with references DSL*. This DSL supports both the domains of the

Listing 1.3. A 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
```

Entities and the Constraints DSLs and enriches them with references between entities. An example sentence of this language is presented in Listing 1.3.

To keep the course pragmatic (and again prevent accidental complexity) we favoured model-based approach to parser generation. Students use the YAJCo [33] model-based parser generator that considers the existing object-oriented model of the Entities with references DSL 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). Moreover, the *meta-model* from previous iterations is *reused* thus mimicking agile evolution of the MDSD solution. Prototype parsers implemented in the previous iterations are discarded, but the meta-model and the generators are still used (with some evolutionary modifications, e.g., students need to add references to the meta-model).

User Interface The last part of CRUD applications that needs to be generated is their user interface (UI). Here again we utilize *language composition*. The UI is supposed to be specified in a *UI specification DSL*. Entity objects are presented to application users in simple tables, each of them with a set of columns corresponding to entity properties. To support creating and editing entity instances, a form has to be specified. Again, for each property, a field in the form is defined. To support individuality and creativity we encourage students to generate desktop, web-based or mobile-based GUI to get additional points if they are skilled enough. This encourages motivation and allows us to show them the full potential of generative programming.

The UI DSL is implemented using the *generic language approach* (called Commercial-Off-The-Shelf by Kosar et al. in [22]). 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 [27]. This popularity is the reason why we believe that generic languages should be a part of an MDSD course.

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 representation. JAXB again enables *meta-model reuse* and what is more important, it is widely used in industry (e.g., in SOAP web services). The language concrete syntax is described also by using popular

Listing 1.4. XML-based user interface specification DSL sentence

```
<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>
```

XML schema to employ another industrially used technology [35]. Listing 1.4 shows a simple user interface specification for the CRUD application with Employee entities using an XML-based notation.

From the viewpoint of code generation and templates we use this iteration to show how the *templates can be reused and maintained* [4, 40]. The UI has to validate user input to avoid violating constraints on entity properties. Here the students have to reuse the constraints validation templates written in the second iteration. This way they can see that a good decomposition of templates can also support template reusability.

5. The Missing Piece: Build Process Visualisation

Our iterative course employs small-scale solutions to solve partial problems of MDSD. As we have discussed, teaching with different types of parsers and generative approaches has multiple advantages to a full-scale MDSD solution. However, tearing the whole solution apart carries the risks of forgetting how those partial solutions fit together into the whole picture. We had experiences that our students understood the advantages of generating software artefacts from a DSL specification, but for many of them the whole solution remained a black box with several tasks that were not connected together in their eyes. To deal with this problem we have implemented a visualisation tool we call *MagsaTool*[2]. The tool is able to load the students' project, illustrate her current progress and execute actions in the students' code by directly manipulating with the visualisation.

For each lesson, *MagsaTool* provides an interactive scheme outlining the build process[3] in the case study. Each scheme is supplemented with a code snippet that implements the build process in Java. Thus students can compare their testing code with the appropriate build code for the lesson. The students cannot change or adapt the schemes, but they can execute actions such as parsing, model composing or code generation directly in the visualisation. Thanks to the visualisation a student can associate the components in the scheme with the artefacts she was (or will be) implementing in the current lesson. The tool also highlights the changes between each of the lessons, thus visually showing to the students what was added to the project and what was discarded.

[2] *Magsa* is an acronym of the course name.

[3] Process of interconnecting all partial solutions to a complete solution.

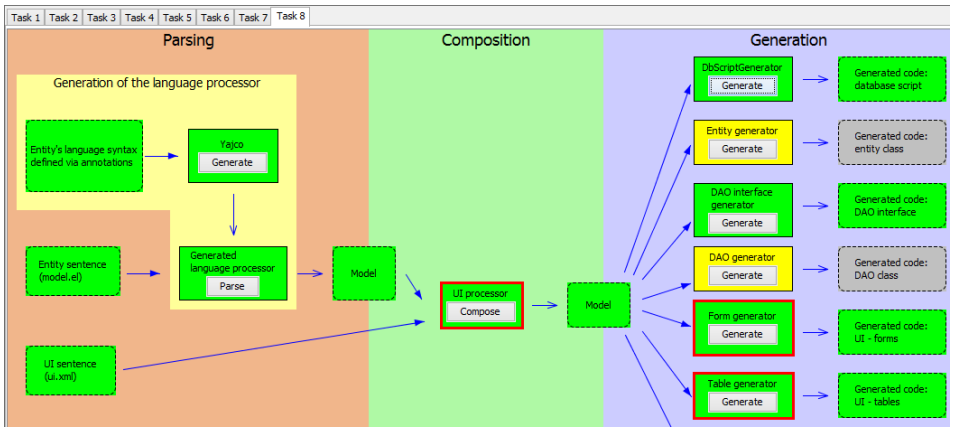

Fig. 3. Case study tooling infrastructure after the last iteration

Figure 3 shows the scheme of the tooling infrastructure for a completed case study. The rectangles with round corners and dashed border are resources (input files, model, output files), the rectangles with solid border and buttons are executable components. Available resources are highlighted by green color (dark gray in black and white print). Executable components are highlighted by yellow (white in black and white print) and they contain buttons with appropriate actions (parse, compose, generate). After run, in case of successful execution, they are highlighted in green. Light gray components are not available (e.g., their input is not available, they were not created yet, etc.). Components or resources can be highlighted in red, if there was any problem in their loading or execution.

The reader may have noticed that some of the generators in Figure 3 have red thick borders. *MagsaTool* highlights new components in the current task with red borders for few seconds to indicate they are the increment of the current iteration. In this case the rest of the components was already implemented in the previous lessons, and only those highlighted generators will be implemented in the current one. Showing the increment of the current lesson helps the students to have a clear idea about what is their task in the current lesson.

The model is put in the middle between the parsing/composition phases and the generation phase to clearly indicate that it is the meeting point of both phases (the interface between them). It shows that if we do not change the model format we can arbitrarily manipulate or substitute the parser (as we do across the iterations), and the generation phase components will remain usable. Another important aspect the students can realize is that although they work on multiple different generators (of course with different outputs) they all work with the same model.

The visualisation of the parser generator in Figure 3 shows to the students the place of parser generator in the MDSD process. In our experience students often get confused in multiple meta layers of generation. The scheme helps them to put it all in the right place. It also makes a distinction between a generated parser and the parser generator[4]. Similar

[4] For some reason students often think that YAJCo itself parses the sentences, they consider the generated parser is a part of YAJCo and it will not work without YAJCo library in the classpath.

problem arises in the second iteration when they implement an internal DSL based on Java. They get confused by using the same language for the sentence definition (an internal DSL), for parsing (expression builder is of course in Java), and also for the generated system (the CRUD applications are generated to Java). Here the visualisation helps again.

Composition with the XML-based language shows to the students that thanks to using Java (or GPL in general) for the model definition we can use not only Java-based parser generators, but all standard tools and technologies on the given platform. In this case JAXB can be used to parse XML documents and to create in-memory objects.

By going through all the schemes visualized by the *MagsaTool* students can put all their work into the big picture. They can understand that although they were implementing 4 different languages with different semantics, they all shared a common MDSD infrastructure (sketched in Figure 2). It is also easier to understand the role of each of the used tools (different parsers, model, etc.) and their interactions.

6. Evaluation

To show the viability of our solution we compare it to multiple similar approaches to teaching MDSD in the Related work section. As an additional support to our claims and to determine the impact of using MDSD in our course, we also administered a survey to the students in our classes. 58 of total 137 students responded the voluntary survey. The 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 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 of the survey addressed the course itself, it's form and the problems that the students might have had during the course. The participants were further interviewed on difficulties faced during the entire course as well as subjective preferences.

6.1. 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 skilled programmers and many of them are focused explicitly on computer networks. 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*.

We interpreted the reasons for such good results[5] are a good motivation and examples from our side, emphasis on industrial practice and the fact, that they could see the whole approach broken into small interconnected parts. This way they understood that MDSD is not only about some "magical" generation of the whole solution without even writing one line of code, but that it can also be applied to little solutions, as it often is in practice. Many of them identified the potential of MDSD in bootstrapping mobile or web applications (e.g. generating the core model classes, basic screens, SQLite database etc.) and one of the students successfully attempted to implement and use such a solution on top of the case study.

The results of the 4th question shows that majority of the students (93%) liked the iterative approach used in our course which is understandable, since this approach was successful in other subjects as well.

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, 14% marked the subject as their favourite in the given semester, other 56% as rather favourite, which is a quite positive result. The fact that this is one of the few practice-oriented subjects in that semester may have improved the results a little bit, however based on the interviews the number of satisfied students was similar.

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 did not report any problems with the course*. A number of students had issues with the techniques used - YAJCo (3 students), annotations (2 students) or Velocity (7 students).

[5] When compared to studies such as http://modeling-languages.com/failed-convince-students-benefits-code-generation/

Although the students had issues, the results in case of the 9th question show that *more than a half* of them (52%) explicitly stated that they *would not change the course materials at all*. The reason may be that although they fought the problems for some time, in the end they always won over them either by themselves or with the help of the instructor and they became satisfied with what they achieved. Thus in the end of the semester, looking back at the whole case study, they perceived the issues as "challenges" rather than as "unsolvable barriers". The presence and help of the instructor directly during the class represents an important factor since the most difficult problems were often solved with his/her help. It is important to help students to solve the problems they fight with for some time so they are not discouraged from continuing the case study.

The results obtained in the open text section and oral interviews showed that in general our students tend to like our approach. Some of them explicitly appreciated the various techniques used and favour the possibility of using the learned techniques in practice.

6.2. Threads to validity

External validity. Although the sample of analysis was sufficient, the validity of this study can be questioned based on the analysis performed only on a single year's classes. A further analysis is needed to find out the impact in the next years and compare the results. Our subjective experience from the past years is that the recent course is more successful than the ones in previous years. In the future we will conduct multiple iterations of the survey to be able to compare the results formally.

Internal validity. Since the survey was voluntary, not all of the students participated, therefore the participants might not represent an average group of students. The non-participants might have seen the course negatively, but were afraid of possible repercussions (although the survey was anonymous, students can be paranoid).

Construct validity. The survey evaluates only subjective opinions and feelings of our students, therefore it can not be used to fully compare our course to different approaches, it only evaluates how our students perceived the course. Since our subject used similar approach for multiple years from the beginning, we do not have comparable empirical data about different courses based on different approaches at the time. However, the comparison can be made based on the existing literature on the topic and is discussed in Section 7.

6.3. Summary

We conclude that our approach to the course has relative success between the students and orientation to the practice has a motivating impact on them. The survey revealed also some problems. Main problems are of technical character, mostly with using YAJCo[6] and the presence of the instructor was a viable factor when solving them. Students also reported issues with tasks definition, the less skilled students considered some of them too hard. The common cause of the task misunderstanding was that although the tasks were

[6] Especially because of error reporting, YAJCo generates JavaCC specification that reports errors in its own vocabulary. Mapping the JavaCC error report to YAJCo meta-model is a hard task even for a skilled YAJCo user.

explained in the class, some of the students did not listen and started to work on them at home, without the instructor immediate feedback. Therefore we recommend to encourage the students to work directly in the class to help them to solve possible issues.

7. Related Work

The motivation for teaching MDSD at our university is based on its promises of narrowing the semantic gap between the problem and solution domains. Selic [37] 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 proclaim the benefits of MDSD on one hand, but on the other hand state that MDSD is given little attention in education (e.g., an early work by Cowling [6]). The main problem with MDSD teaching at our university is that myths discussed in Section 1 were and still are strongly rooted among our students. Although there are numerous other challenges in MDSD and MDSD teaching (e.g., work by France et al. [12]) we faced the problem of students' scepticism against MDSD. Most of available scientific works were just proofs of those myths to our students, because they usually deal with 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 agile methodologies, etc.).

MDSD teaching approach Considering the taught principles we explain the MDSD topic from the viewpoint of the language-oriented programming (details of the course from this perspective can be found in [32]). Although this viewpoint covers basically the same challenges and benefits as rather "classic" MDSD teaching approaches, such as the one by Clarke et al. [5], our approach is based on multiple interchangeable tools and techniques. We also decided to focus more on the topic of formal languages, since currently there is an ubiquitous need for developing and working with little languages in the industry (especially configuration languages [27]). As described by Picek and Starhonja [31], MDSD has more potential when using formal languages, especially DSLs, in opposed to graphical languages such as UML [14].

Tools and models Problem with tools used in MDSD teaching was discussed in multiple scientific works. There are cases in which teachers chose complex MDSD tools without reporting any significant problems with students using them. For example, Tekinerdogan [39] and Clarke et al. [5] used Eclipse modelling Project (EMP) tools, Pareto [29] used Microsoft DSL toolkit. However, there are some reports of students having problems with working with such complex tools. For example, Batory et al. [3] tried to use the Eclipse modelling 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 challenge the myths about the solely large scale MDSD application and the need of massive tool support.

There are also works that use tools and models developed specifically for teaching MDSD, such as the MIST tool by Dimitrieski et al. [9]. Compared to our approach, Dimitrieski et al. use an EER DSL to express the domain. EER is similar to entity-relationship model, which their students are familiar with, therefore they recognize the notation. To be able to generate SQL code, application code or GUI, an explicit transformation has to be specified (mapping between EER and the other formats). Considering this fact, our entity DSL notation is more abstract, although it is not as common and recognizable as EER would be and it does not provide as many possibilities for transformation. Instead of focusing on existing entity-relationship model our students develop their own little language. We decided not to use an existing modelling language because we suspected that then some of them might not understand that MDSD can be used for an arbitrary problem an in turn think that we are speaking about compilers design, etc.

As opposed to an explicit specification of platform-specific details in the model, we used platform-independent models as defined by Djukić et al. in [10] to make the students implement as much as possible in the generator code. The only mapping that we use in the final solution is between the model and the UI. Here we agree with Dimitrieski et al. [9] that the mapping from entities and properties to screens and components has to be explicitly specified in the model in order to show to the students that the UI representation specification can (and should) be separated from the core model to ensure product adaptability.

Schmidt et al. [36] identified three approaches to MDSD teaching: *(i) purely theoretical approach* that focuses on theoretical knowledge and neglects the practical exercise of the MDSD principles by students themselves; *(ii) tool-supported approach* is a teaching approach that uses a single complex MDSD tool (e.g., EMF in [5]); and *(iii) practical approach* that focuses on underlying concepts rather than the use of a concrete tool. Schmidt et al. [36] use the practical approach in which they ask students to implement the generator tool by themselves. They favoured this approach over 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.

Domain selection As many projects showed, careful consideration of the domain of the course project is very important, especially in the fields where formal methods [20], [41] and specifications [19] [18] are used. For example, Mosterman [26] uses the domain of embedded systems, Clarke et al. [5] use the domain of communication services and Batory et al. [3] let the students choose a domain of their interest. We think CRUD is the best choice for us for the reasons we explained in section 4.2. 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.

Teaching approach From the viewpoint of the teaching approach, most of the articles report using classic development with a single iteration (e.g., Clarke et al. [5]). 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. [36]. 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. Thus, each iteration explains a different topic. In our approach the iterations' character is quite different; in each iteration we reiterate the whole MDSD process, only with different tools and techniques. We believe that repeating the same concept with different implementation details helps a deep understanding of the basic MDSD principles.

Evaluation Barišić et al. [2] present a controlled experiment that evaluates usability of a DSL in comparison with GPL. The experiment was conducted with two groups of physicists using a Pheasant DSL for High Energy Physics, and a GPL solution based on C++ and BEE library. The results showed that DSL had significantly better results in effectiveness, efficiency, and confidence. Kosar et al. [23] use cognitive dimensions framework to evaluate the difference between program understanding using a DSL (XAML) and using a GPL (C# Forms). They confirm the better understandability of the DSL over GPL. Moreover, they use cognitive dimensions framework to analyse the study results of the used languages with respect to cognitive dimensions (Closeness of mapping, Viscosity, etc.). In our work we used a simple questionnaire combined with interviews to get feedback from our students. While the results were quite optimistic, since we did not conduct a controlled experiment, they are also quite disputable. In future work we want to design a controlled experiment to compare our approach to a "conventional" teaching approach using an integrated MDSD solution. Also, as Kosar et al. [23], we should be able to better analyze the reasons for the success of our approach from aspects of cognitive dimensions.

8. 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 in a way that the students would understand the high applicability of small scale MDSD in practice. 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 an inspiration when adapting the model-driven approach in an agile development of a software project. Although the presented approach is not a silver bullet, in our experience it helps us to engage our students in MDSD topic and to provide them with necessary knowledge required for MDSD adoption in practice. The guidelines which are the results of this work can be beneficial for all the MDSD teachers that meet with the myths presented in Section 2 spread among their students.

In future research we plan to improve the subject based on students' feedback: improve the task formulations, start using Maven to solve IDE compatibility problems and the next step is to answer the problem of high work load by separating the subject into two new subjects: "Modelling and Generating" and "Domain-Specific Languages".

Acknowledgments. 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. Ambler, S.: Agile model driven development is good enough. IEEE Software 20(5), 71–73 (Sep 2003)
2. Barišić, A., Amaral, V., Goulão, M., Barroca, B.: Quality in Use of Domain-specific Languages: A Case Study. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools. pp. 65–72. PLATEAU '11, ACM, New York, NY, USA (2011)
3. Batory, D.S., Latimer, E., Azanza, M.: Teaching model driven engineering from a relational database perspective. In: MODELS 2013: 16th International Conference Model-Driven Engineering Languages and Systems. pp. 121–137 (2013)
4. Christou, M., Flouri, T., Iliopoulos, C., Janoušek, J., Melichar, B., Pissis, S., Žďárek, J.: Tree template matching in unranked ordered trees. Journal of Discrete Algorithms 20, 51–60 (2013)
5. Clarke, P.J., Wu, Y., Allen, A.A., King, T.M.: Experiences of Teaching Model-Driven Engineering in a Software Design Course. In: MODELS 2009: 12th International Conference on Model Driven Engineering Languages and Systems (2009)
6. Cowling, A.J.: Modelling: a neglected feature in the software engineering curriculum. In: CSEE&T 2003: 16th Conference on Software Engineering Education and Training. pp. 206–215 (March 2003)
7. Dejanović, I., Milosavljević, G., Perišić, B., Tumbas, M.: A Domain-Specific Language for Defining Static Structure of Database Applications. Computer Science and Information Systems 7(3), 409–440 (2010)
8. Demir, A.: Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. In: MBD-MOMPES'06: Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software. pp. 75–83 (2006)
9. Dimitrieski, V., Ćeliković, M., Aleksic, S., Ristić, S., Luković, I.: Extended entity-relationship approach in a multi-paradigm information system modeling tool. In: FedCSIS 2014: Federated Conference on Computer Science and Information Systems. vol. 2, pp. 1611–1620 (2014)
10. Djukić, V., Luković, I., Popović, A., Ivančević, V.: Model execution: An approach based on extending domain-specific modeling with action reports. Computer Science and Information Systems 10(4), 1585–1620 (2013)
11. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional (2010)
12. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: FOSE'07: Future of Software Engineering at ICSE. pp. 37–54 (2007)
13. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
14. Havlice, Z.: Auto-reflexive software architecture with layer of knowledge based on uml models. International Review on Computers and Software (IRECOS) 8(8) (2013), `http://goo.gl/gakEMI`
15. Hunt, A., Thomas, D.: The pragmatic programmer: from journeyman to master. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
16. Hutchinson, J., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Science of Computer Programming 89, Part B(0), 144–161 (2014)

17. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
18. Kollár, J., Halupka, I.: Role of Patterns in Automated Task-Driven Grammar Refactoring. In: SLATE 2013: 2nd Symposium on Languages, Applications and Technologies. pp. 171–186 (2013)
19. Kollár, J., Halupka, I., Pietriková, E.: A task-driven grammar refactoring algorithm. Acta Polytechnica 52(5), 51–57 (2012)
20. Korečko, v., Sorád, J., Dudláková, Z., Sobota, B.: A toolset for support of teaching formal software development. In: Software Engineering and Formal Methods, Lecture Notes in Computer Science, vol. 8702, pp. 278–283. Springer International Publishing (2014)
21. Kos, T., Kosar, T., Knez, J., Mernik, M.: From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. Computer Science and Information Systems 8(2), 361–378 (2011)
22. Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 50(5), 390–405 (Apr 2008)
23. Kosar, T., Oliveira, N., Mernik, M., Pereira, V.J.M., Črepinšek, M., da Cruz, D., Henriques, R.P.: Comparing general-purpose and domain-specific languages: An empirical study. Computer Science and Information Systems 7(2), 247–264 (2010)
24. Macek, O., Richta, K.: Application and Relational Database Co-Refactoring. Computer Science and Information Systems 11(2), 503–524 (2014)
25. Mernik, M.: An object-oriented approach to language compositions for software language engineering. Journal of Systems and Software 86(9), 2451–2464 (2013)
26. Mosterman, P.: Automatic Code Generation: Facilitating New Teaching Opportunities in Engineering Education. In: 36th Annual Frontiers in Education Conference. pp. 1–6 (Oct 2006)
27. Nosáľ, M., Porubän, J.: XML to annotations mapping definition with patterns. Computer Science and Information Systems 11(4), 1455–1477 (2014)
28. Ober, I., Abou Dib, A., Féraud, L., Percebois, C.: Towards Interoperability in Component Based Development with a Family of DSLs. In: Software Architecture, Lecture Notes in Computer Science, vol. 5292, pp. 148–163. Springer Berlin Heidelberg (2008)
29. Pareto, L.: Teaching Domain Specific Modeling. In: 3rd Educators symposium at MODELS 2007. p. 7 (2007)
30. Perišić, B., Milosavljević, G., Dejanović, I., Milosavljević, B.: UML profile for specifying user interfaces of business applications. Computer Science and Information Systems 8(2), 405–426 (2011)
31. Picek, R., Strahonja, V.: Model driven development - future or failure of software development? In: IIS'07: 18th International Conference on Information and Intelligent Systems. pp. 407–413 (2007)
32. Porubän, J., Bačíková, M., Chodarev, S., Nosáľ, M.: Pragmatic model-driven software development from the viewpoint of a programmer: Teaching experience. In: FedCSIS 2014: Federated Conference on Computer Science and Information Systems. pp. 1647–1656 (Sept 2014)
33. Porubän, J., Forgáč, M., Sabo, M., Běhálek, M.: Annotation Based Parser Generator. Computer Science and Information Systems 7(2), 291–307 (Apr 2010)
34. Porubän, J., Nosáľ, M.: Practical Experience with Task-driven Case Studies. In: ICETA 2014: IEEE International Conference on Emerging eLearning Technologies and Applications. vol. 11, pp. 367–372. IEEE (2014)
35. Pušnik, M., Heričko, M., Budimac, Z., Šumak, B.: XML Schema metrics for quality evaluation. Computer Science and Information Systems 11(4), 1271–1289 (Oct 2014)
36. Schmidt, A., Kimmig, D., Bittner, K., Dickerhof, M.: Teaching Model-Driven Software Development: Revealing the "Great Miracle" of Code Generation to Students. In: Sixteenth Australasian Computing Education Conference (ACE2014). CRPIT, vol. 148, pp. 97–104. ACS, Auckland, New Zealand (2014)

37. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (Sep 2003)
38. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
39. Tekinerdogan, B.: Experiences in teaching a graduate course on model-driven software development. Computer Science Education 21(4), 363–387 (2011)
40. Černý, T., Macik, M., Donahoo, M.J., Janousek, J.: Efficient description and cache performance in aspect-oriented user interface design. In: FedCSIS 2014: Federated Conference on Computer Science and Information Systems. vol. 2, pp. 1667–1676 (2014)
41. Šimoňák, S.: Verification of communication protocols based on formal methods integration. Acta Polytechnica Hungarica 9(4), 117–128 (2012), http://www.uni-obuda.hu/journal/Simonak_36.pdf
42. Whittle, J., Hutchinson, J.: Mismatches between industry practice and teaching of model-driven software development. In: Models in Software Engineering, vol. 7167, pp. 40–47. Springer Berlin Heidelberg (2012)
43. Živanov, Ž., Rakić, P., Hajduković, M.: Using Code Generation Approach in Developing Kiosk Applications. Computer Science and Information Systems 5(1), 41–59 (2008)

Jaroslav Porubän is Associate professor and the Head of Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is a member of the Department of Computers and Informatics at Technical University of Košice. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain specific languages and computer language composition and evolution.

Michaela Bačíková is Assistant professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. She received her PhD. in Computer Science in 2014 for her work in the field of domain analysis of graphical user interfaces. Currently her research focuses on domain-specific languages, domain analysis, graphical user interfaces, usability and user experience.

Sergej Chodarev is Assistant professor at the Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his PhD. in Computer Science in 2012. The main areas of his current research are design and implementation of domain specific languages, metaprogramming and user interfaces.

Milan Nosáľ is PhD student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Computer Science in 2011 for his work in the field of configuration formats and attribute-oriented programming. Currently his research focuses on attribute-oriented programming, program comprehension and projectional programming.

Received: January 7, 2015; Accepted: June 4, 2015.