Teaching Domain-Specific Language Engineering and Model-Driven Software Development

A competence-oriented approach

Volkhard Pfeiffer

Department of Electrical Engineering and Computer Science
Coburg University of Applied Sciences and Arts
PO 1652, 96406 Coburg Germany
volkhard.pfeiffer@hs-coburg.de

Abstract. Teaching and learning domain-specific language (DSL) engineering and model-driven software development (MDSD) concepts are difficult tasks: either it requires a deep understanding of the nature of a domain, students lack it in general or students are exercising only single technical aspects of MDSD, so that they don't see the whole picture and are lost in the model-driven and tool "jungle".

This paper explains a competence-oriented approach for model-driven software development course design to reduce the above learning difficulties. The main idea is first to define the course competencies students should have in a precise manner and second to choose an "appropriate" didactic method for each required competency. Two didactic examples are presented: Peer Instructions for MDSD fundamentals and a comprehensive MDSD software project for DSL and transformation competencies, in which students need to develop a complete workflow system for examination regulation issues. At the end we discuss the overall experience with this approach and with the current course settings.

Keywords: competency, subject-matter didactic, didactic methods, domain-specific language, model-driven software development

1 Introduction

Domain-specific languages in the context of model-driven software development have become increasingly popular in the software industry and are currently achieving the plateau of productivity of the technology hype cycle. Academically MDSD courses have been integrated in software engineering curricula at many universities. The following subject-matter didactics key questions arise for our course:

- What kind of DSL- and MDSD-competencies should a MDSD course address?
- What is the best way of learning the potential of model-driven technologies?
- How do we focus teaching on DSL- and MDSD concepts rather than on (different) tools?

The course (optional module, 4 contact hours, 6 credits) is part of the graduate curriculum of a master degree program in Computer Science at a University of Applied Sciences in Germany with a more applied research orientation; in contrast to traditional universities with a stronger theoretical academic research focus. Prerequisites are good knowledge of software engineering, software architecture and programming languages. It is assumed that students have already gained first-hand experience with the execution of traditional and agile process models for larger software projects.

The module is designed according to the following principles:

- *Focus on Languages* Modeling subjects are commonly not students' favorites in contrast to programming languages and software design; in particular to our students with a more practical orientation. We still want to improve their modeling skills. A language viewpoint therefore should stimulate the learning motivation.
- *Foster High-Level Abstractions and generative programming* We want to achieve acceptance of MDSD – similar to the acceptance of compilers. Hence, Software generation as an example for automating software development is an excellent use case which demonstrates the MDSD potential and it should also increase acceptance. Therefore we have to design practical exercises in a way that students are enabled to define high level abstractions as well as to implement code generators.
- *Exercise from a system viewpoint* Neither teaching DSL designs alone nor teaching model transformations alone is sufficient. Instead, students have to synthesize these techniques and have to implement a complete system to realize the advantages.

The rest of the paper outlines our approach to answer the key questions: §2 explains some of the intended competencies in a precise manner. §3 presents two didactic examples to achieve these competencies. §4 discusses our experiences with the didactic approach from both teacher and student perspective.

2 Competencies – a pragmatic view

2.1 Definitions and Terms

The term "competency" is one of the most popular and most confusing terms with different definitions and meanings used in education. A most cited definition is Weinert [11] p. 46 who defines competency as the "existence of learnable cognitive abilities and skills which are needed for problem solving as well as the associated motivational, volitional and social capabilities and skills which are needed for successful and responsible problem solving in variable situations." I.e. technical knowledge (often referred as factual knowledge) as well as "soft skills" (non-technical knowledge) are competency ingredients.

In this paper we do not further discuss this (and other) competency definitions. Instead, we describe competencies by learning outcomes, which are defined according to [4] p. 10 as "statements of what the individual knows, understands and is able to do on completion of a learning process". Following Weinert's definition we classify these learning outcomes into technical and non-technical categories.

2.2 Technical Learning Outcomes

The course covers MDSD terminology, meta-modeling, model-to-model and model-to-text transformations, internal and external domain specific languages and model validation. Model management is skipped due to time constraints.

The addressed technical learning outcomes are specified in detail; a subset is listed in Table 1[1]. Each learning outcome has an associated level of mastery according to the Anderson and Krathwohl (AKT) learning objective taxonomy [1]. Although this classification is subjective, it supports course design (see [5] for further discussions).

Table 1. Detailed technical Learning Outcomes (subset)

Students should be able to		***AKT Level[2]***
1.	**MDSD Basics**	
	explain the MDSD terminology in their own words	2
	classify the different MDSD approaches	2
2.	**Meta-Modeling**	
	explain the UML/Meta Object Facility (MOF) core package and the UML extension mechanism in their own words	2
	assign an (UML) model to the correct meta model hierarchy	2
	explain the Ecore meta model in their own words using an example	2
	create an Ecore meta model for a given textual description of an application domain	6
3.	**External DSL**	
	develop and implement a technical[3] DSL in arbitrary textual notation for a technical domain using parser generator tools	6
	develop and implement a business[3] DSL in arbitrary textual notation for a given application domain using parser generator tools	6
	validate the usability of a business DSL for the DSL user	5
4.	**Model-to-Text Transformation**	
	decide which parts of a system can be implemented by existing technologies (instead of generating) for a textual requirement specification	5
	decide which parts of a system can be generated for a given software architecture	5
	apply “best practices” generation patterns	3
	implement and test a generator for a given non-trivial model representation using template-engines	6

Learning outcomes listed under Table 1 3. have certain implications: in order to find a compromise between tool learning curve and DSL expressivity these competencies are restricted to textual concrete syntax and parser generator tools. This might have a major impact especially on the definition of business DSL’s.

[1] Course topics not listed (e.g. model-to-model transformations) are handled similarly.
[2] Level 2 means „understand“, Level 3 “apply”, Level 5 „evaluate“, Level 6 „create“
[3] for further discussions see [10] p. 26

2.3 Non-technical Learning Outcomes

Soft-skill recommendations particularly relevant for software engineers exist in a fairly different level of description (e.g. [2]). The detailed non-technical competencies proposed in [8] are also required to master MDSD projects. A few of them are fostered in this course explicitly (s. Table 2).

Table 2. Detailed non-technical Learning Outcomes (subset)

Students should be able to		
1.	**Think Abstractly**	
	abstract context and requirements for a given non-trivial domain problem description independent from how they are implemented	3
	abstract system behavior and structure of the application independent from implementation platform	3
	identify technical platform implementation aspects	3
	identify boiler-plate code and syntactical noise	3
	evaluate if the abstraction is meaningful for the given task	5
2.	**Self-reflect**	
	reflect on their capabilities according to MDSD activities, engineering guidelines and roles (e.g. language designer, software architect, modeling expert, generator expert)	5

3 Didactic Approaches

This section explains some didactic examples used in the current course. A specific teaching and exercise format is chosen depending on the required learning outcomes.

3.1 MDSD Meta-Modeling

Peer Instructions [6] and/or Just-in-Time Teaching (JiTT) [7] are useful teaching methods for MDSD terminology and classification: a multiple choice concept question is posed and students vote by using a clicker-response system. If a large number of answers are wrong, students are asked to justify their answer with their neighbor. After the discussions the class is polled again on the same question. A typical question discusses the quality of a given (Ecore) meta-model (see Fig. 1). We frequently begin a lecture with a peer instruction to summarize the topics of the last lecture(s).

Select correct answers:

- finalstate cardinality wrong
- incoming transition reference must be containment
- transition meta class correct
- a meta class is missing

Fig. 1. Multiple choice answers for a given Finite-State-Machine Meta model question

3.2 External DSL and Model-to-Text Transformation

In order to acquire DSL and transformation competencies and according to our system viewpoint project work is set up to implement a complete software system with the following parameters and settings:

1. *Domain* A software company is developing examination regulation software systems for university and college customers. The system has typical course administration and information requirements: lectures enter/edit module grades; students query their study progress and register for examinations. The system should also support different types of verification e.g. check of all prerequisites for a module examination registration.
2. *Architecture* Only the architecture layering is pre-defined (see Fig. 2).
3. *DSL* Two kind of DSL's and generators have to be designed and implemented:
   (a) an entity DSL for the persistence layer as an example of a technical DSL
   (b) high abstraction examination regulation DSL as an example of a business DSL, which should enable the generation of business layer and presentation layer parts.

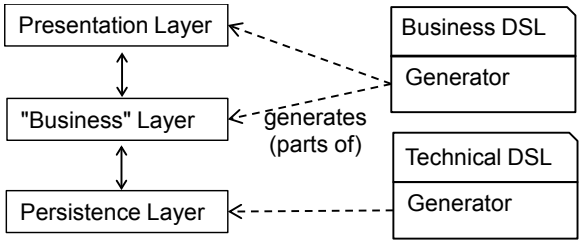

Fig. 2. High level requirements: Multi-Tier Architecture and two DSL's

4. *Implementation technologies* Students select individually the implementation technologies and frameworks. Typically a web-based architecture is designed.
5. *Process model* The iterative method SCRUM is applied due to the fact that our students are already familiar with SCRUM.
6. *Project Size* In order to focus each team member on MDSD activities students are divided into small teams of 3 individuals only. All projects run simultaneously to accomplish the same task.
7. *Tool* Xtext/Xtend [12] is the tool DSL/Generator infrastructure.

These settings have advantages and risks:

- Both DSL domains (examination regulations and database persistence technologies) are well-known domains to our students. Thus, the domain learning curve is minimized.
- A variety of entity DSL examples in different syntax styles have been discussed in the literature (e.g. [9][12]). Hence, students are learning their first DSL design by example as well as the Xtext tool. One of the first iterations starts with this entity

DSL - a good preparation for the business DSL design implemented in subsequent iterations.

- A major risk is that the effort for implementing architecture parts (not directly or indirectly related to MDSD technologies and activities) is too high. As a consequence students
  - are only allowed to select technologies and frameworks which they are familiar with
  - have to reuse existing technologies as much as possible
  - have to design a "simple" software architecture (no over engineering)

  In our role as a coach we review thoroughly the proposed software architecture considering these guidelines.
- We weekly check the students' iteration proposals in order to avoid iteration planning errors (e.g. DSL definition iteration before finishing a domain analysis).
- Xtext is an example of a "grammarware" tool (cp. [3] p. 15). Hence, it is even more important to focus students' language design on abstract syntax development using meta-modeling.
- Soft-skills will be fostered only to some extent due to the fact that team size is limited to 3 individuals.

The software project is embedded in the exercise schedule as depicted in Fig. 3:

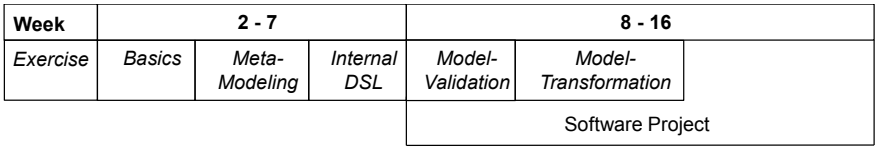

Fig. 3. Rough exercise time schedule: (bi-) weekly exercises and the software project

4 Experiences

This section evaluates the didactic approach from both teacher and student perspective. It is based on a 6 year MDSD teaching experience and regularly student surveys among all attendees (16 on average).

1. Detailed learning outcomes support course design

Our approach for specifying learning outcomes is the process of refining the module learning objectives in a detailed manner – similar to requirement analysis activities applied for the course requirements itself. Hence, module handbook learning outcomes are described much coarser and the number is commonly restricted to the top five to six. The approach has two advantages: on the one side useless knowledge will be omitted. On the other side all required competencies are clearly specified in detail. This level of detail enables a setup of "appropriate" didactic methods as well as design competence oriented examinations and assessments.

2. Peer Instructions appropriate for (low-) level 2 learning outcomes

Most of the addressed level 2 learning outcomes can be assessed by well-thought-out questions. Peer Instructions based on these questions enable the measurement of

learning improvements: the percentage of the correct answers, after the discussions, increases significantly (e.g. by factor 2).

3. Choosing the right context is a key factor for understanding and acceptance

The addressed model-to-text transformation as well as the non-technical competencies require a non-trivial exercise task. The assigned project work fulfills these requirements and supports learning: At the beginning of the course less than 10% of all students have ever heard of MDSD technologies. During the project phase students were realizing the benefits of high abstractions: essential system parts have been generated out of "their" own DSL: an artifact is shown in Fig. 4. At the end of the project at least 70% of all students claimed that they would use specific MDSD technologies in industry. This implies that exercising these techniques in the same context allows students to gain a much better understanding of the various MDSD technologies.

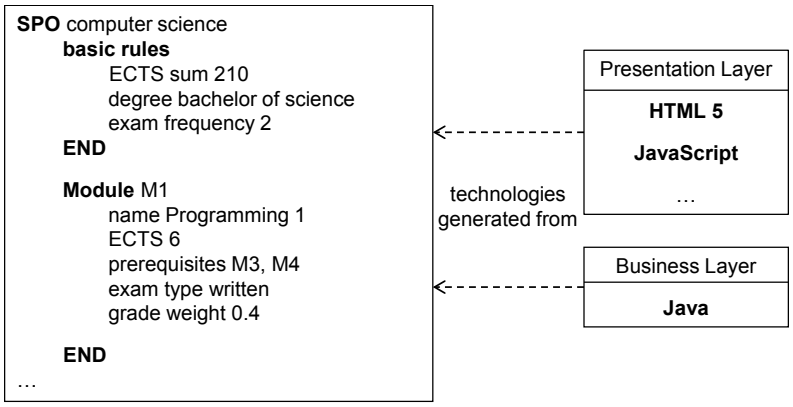

Fig. 4. Concrete DSL example artifact for examination regulation system

4. Software Project is students' favorite and motivation is high

Students complained about considerable project effort. Nevertheless, it showed the best evaluation results. Some teams delivered more functionality than requested. DSL design, learning template generation patterns were rated as "most interesting".

5. Plenty of coaching is required for design decisions and planning issues

A major difficulty for students was to determine whether an abstraction is part of the platform or part of the DSL. In addition, we permanently have to review all planning activities. This indicates the difficulty in tailoring process models to MDSD.

6. Xtext/Xtend learning curve acceptable for an eight week software project

Our experience is that most students learn Xtext basics in ~2 days. Hence, students make quick progress. The Xtend learning curve takes longer (~5 days).

5 Summary and Outlook

In this paper we have presented our didactic methodology and some concrete didactic examples in teaching and learning model-driven software development. We follow a

top-down approach where we first define competencies by learning outcomes in a precise manner. The proposed level of detail is intended as a tool for course design. The selected learning outcomes of the course are a personal decision and might differ from MDSD to MDSD course. Our MDSD course illustrates several MDSD techniques and activities by using e.g. Peer Instructions and JiTT, different exercises and a comprehensive software project.

We intend to evolve the teaching format with respect to the following aspects: In the current course the designed examination and regulation DSL is not validated by a "real" customer. Simulating a real customer should improve the concrete syntax style and should also foster communication skills. Additionally, model-to-model transformations are covered, but not exercised entirely due to time constraints in the current setting, which is a major limitation. In order to acquire better acceptance for model-to-model transformations the software project task may be extended by integrating a reasonable model-to-model transformation use case.

Acknowledgments. The work is part of the project EVELIN funded by the German Ministry of Education and Research under grant no. 01PL12022A.

References

1. Anderson L. W., Krathwohl D.R. (Eds.): A Taxonomy for Learning, Teaching, and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives. Abridged Edition. New York: Longman (2001)
2. Bourque P., Fairley R.: SWEBOK V3.0 – Guide to Software Engineering Body of Knowledge. https://www.computer.org/web/swebok/index
3. Brambilla M., Cabot J., Wimmer M.: Model-Driven Software Engineering in Practice. Morgan (2012)
4. ECTS Users' Guide. http://ec.europa.eu/education/library/publications/2015/ects-users-guide_en.pdf
5. Fuller U. et al.: Developing a Computer Science-specific Learning Taxonomy. ITiCSE-WGR 07 Working group reports on ITiCSE on Innovation and Technology. In: Computer Science Education (2008)
6. Mazur E.: Peer Instruction: A User's Manual. Upper Saddle River. New York: Prentice-Hall (1997)
7. Novak G., Gavrin A., Christian W., Patterson E.: Just-In-Time Teaching: Blending Active Learning with Web Technology. Upper Saddle River, NJ: Benjamin Cummings (1999)
8. Sedelmaier Y., Landes D.: A Software Engineering Body of Skills. In: Global Engineering Education Conference (EDUCON), pp. 395–401. IEEE (2014)
9. Vlisser E.: WebDSL: A Case Study in Domain-Specific Language Engineering. TU Delft Report TUD-SERG-2008-023 (2008)
10. Voelter M.: DSL Engineering – Designing, Implementing and Using Domain Specific Languages. CreateSpace Independent Publishing Platform. (2013) http://dslbook.org
11. Weinert F.E.: Concept of Competence: A Conceptual Clarification. In: Rychen, D., Salganik, L. (eds.): Defining and Selecting Key Competences. p. 46. Seattle: WA: Hogrefe &Huber (2001)
12. Xtext Language Engineering for Everyone. https://eclipse.org/Xtext