Teaching MPS: Experiences from Industry and Academia

Mikhail Barash and Václav Pech

Abstract In this chapter, we present our experience on teaching MPS to industry professionals and university students. JetBrains has run its commercial courses on MPS both online and on-site for 19 different customers coming from 12 countries over the past 5 years, for some repetitively. The participants typically started the courses with no or little language engineering background and no experience with MPS. Although language design with MPS was the main objective of the courses, some general language engineering topics were also covered, and motivation into the domain was provided. We also report our teaching experience of a course on the implementation of domain-specific languages (DSL), given at universities in Finland, Norway, and Canada. The course, aimed at a broad audience of computer science students, covers DSL design principles and implementation techniques using three text-based language workbenches (Eclipse Xtext, Spoofax, Rascal MPL), prior to introducing MPS. By the time MPS is discussed, students have an understanding of what language implementation comprises of and are somewhat fluent in language engineering terminology. This enabled us to focus on language implementation techniques that are distinctive in MPS. We discuss in this chapter the experience we gained during these courses and the evolution of the educational approach.

M. Barash (✉)
Bergen Language Design Laboratory, University of Bergen, Bergen, Norway
e-mail: mikhail.barash@uib.no

V. Pech
JetBrains s.r.o., Praha, Czech Republic
e-mail: vaclav.pech@jetbrains.com

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_10

1 Teaching MPS in Industry: Experience of JetBrains

1.1 History and Motivation

JetBrains has been offering courses for industry professionals since 2015 [32]. Initially, these courses were made on an ad hoc basis as clients approached JetBrains and asked for training individually. The content and structure of the courses stabilized as we ran a couple of these early courses, which allowed us to put out an official offer of two courses in 2017—the *Introductory MPS* course and the *Advanced MPS course*. The courses were organized either at the JetBrains office in Prague or at the client premises. More than 130 people have passed these courses during the first 2 years. The two courses followed one after the other, but it was not mandatory that the students must pass the Introductory course before they sign up for the Advanced one. This demanded the lectures of the Advanced course to invest some initial time into bringing everybody up to the same level, until the online version of the Introductory course became available in 2019 and allowed the students to self-study the materials before joining the Advanced course.

1.2 Going Online

On-site courses proved not to be practical for all clients.

- *Individuals*, for example, found it inconvenient to wait for a large enough group of students to form for a course to start.
- The *travel costs* (both time and money), on the other hand, were increasing the expenses incurred by clients outside of Europe.
- *Students and academy workers* sometimes could not afford the price associated with the courses.
- Innovators, who only wanted to test the waters and see if MPS is an answer to their questions, did not want to *invest the time* necessary to organize and pass a traditional course.

This led us to considering online variants of the courses. Both Introductory and Advanced courses have been migrated into the JetBrains online learning platform Stepik (http://stepik.org). The contents of the online courses are identical to the contents of the on-site courses. However, there have been several changes required:

- Each course is split gradually into *modules*, *lessons*, and *steps*. A typical lesson takes about 30 min to complete and contains between 5 and 15 steps.
- The trainer's explanation is replaced with pre-shot screencasts, typically 3–8 min long.
- The theory is then explained again statically using text and images. Step-by-step tutorials guide the students through the practical aspect of the chapter.

- The students can leave comments for the trainer whenever they get stuck or find some information puzzling or incomplete. This proved to be an efficient mechanism to improve the quality of the course over time.
- The instructions of the online tutorials must be more detailed compared to the instructions in printed tutorials, since the cost of a student making a mistake is much higher in the online course. For the same reason, the common traps have to be identified, and fallback instructions should be provided to help students to avoid getting stuck. This is where comments proved to be indispensable.
- Independent exercises provide an example solution in the following steps and rely on the student not to jump ahead.
- Quizzes with test questions are scattered throughout the course to ensure the students understand the topics.
- Live online discussions with the trainer can be organized to give the students an opportunity to ask for additional explanation, get the trainer's opinion on applicability of MPS to their particular problem domain, or discuss MPS beyond the scope of the course.

Elementary Course In parallel to these online courses, the free online educational content has evolved into a course of its own right. This course was named *Elementary*. Although not as in-depth as the Introductory course and lacking the possibility to chat with the teacher, the Elementary course provides a zero-ceremony option at a zero cost. This clearly rings a bell with many MPS fans. To date, this is the most popular online course of these three.

Online courses proved to require smaller organizational ceremony compared to on-site courses. Especially, the number of individual participants has increased, thanks to the online option. During the first 18 months, 350 students have passed any of the online MPS courses.

1.3 The Participants

The students of the courses can be categorized into several groups (listed roughly by the numbers in our courses).

Group 1. Language engineering experts who want to expand their expertise into a new territory. They typically join the courses with ideas of their next language already sketched in their heads and want to discuss them further.

Group 2. Software developers who understand programming well and want to learn more about language engineering and MPS in order to use it on their next project. They frequently join the courses skeptical about introducing a new piece of technology into the project and are to a great level unaware of the possibilities of today's DSLs.

Group 3. Project managers and architects who want to understand the field of DSLs as well as the MPS technology itself in order to assess it and weigh its benefits. Their programming skills are frequently a bit rusty; on the other hand, they do not need to go in depth into all the exercises in order to get the information they need about the technology.

Group 4. Domain experts who will eventually be using the languages designed in MPS. They typically lack programming skills beyond basic syntax of some scripting languages.

Group 5. Technically savvy business owners who want to see if the technology fits their idea of a new product.

While the Introductory course is relevant to all five groups, only Groups 1 and 2 should consider the Advanced course.

- Groups 1, 3, and 5 are usually self-motivated and enthusiastically pass the courses, optionally skipping parts that they think are not relevant to their business at the moment. During live interactions, they tend to focus on their problem domain.
- Group 2 needs to be first sold on the idea of DSLs and their relevance to their current activities. Practical examples of integrating MPS into the typical technology stacks should be provided to relieve them of some of their doubts. If this succeeds, they normally make fast progress and successfully pass the courses.
- Group 4 is usually the biggest challenge, especially if they are mixed in a group of people belonging to some of the other four categories. Their lack of solid programming skills slows them, and thus the whole group, down. Additionally, the purpose of using DSLs in their particular case must be explained to them before they gain motivation to go through the exercises. These people rarely sign up for the online courses.

The courses assume no prior knowledge beyond general programming abilities and experience with software development. As MPS is a Java-based system, knowledge of Java programming helps the students make progress faster. A prior knowledge of competing language engineering tools has somewhat ambivalent effect. Since the MPS editor is based on the technology of projectional editing and the language definition uses object-oriented principles instead of grammars and parsers, people versed in parsing must mentally overcome the abstraction gap, which is not always easy, especially when it is combined with an *a priori* skepticism toward the projectional technology. In general, self-motivated students fare much better than students signed up by their employers without first highlighting the relevance of the technology to their job.

1.4 Topics Covered in the Courses

Introductory Course The Introductory course (see Table 1) focuses on the fundamentals of the MPS technology as well as the language engineering domain in general. On top of the general description of the technology and motivational

Table 1 Outline of the Introductory MPS course given by JetBrains. The course is meant for participants with no or minimal knowledge of both MPS and language engineering and is focused on the practical application of MPS, based on two major examples and a few shorter exercises. The duration of the course is 16 h

Language engineering: fundamentals of language design
Examples of existing MPS usages
Understanding the MPS user interface
MPS project structure and dependencies
Commanding the projectional editor
Introduction into the aspects of language definition
Effective navigation around code
Constraints and scoping
Language manipulation API: the SModel language
BaseLanguage and its extensions
Modularizing the editor definition and making the editor fluent
Quotations and antiquotations
Generator templates and macros
Tooling for language engineering

examples, it explores the essential aspects of language definition—STRUCTURE, EDITOR, and GENERATOR. It also touches on some of the other aspects, like CONSTRAINTS and INTENTIONS, to make the exercises more realistic.

The Introductory course aims at making the students aware of the technology and being able to create simple languages with it. It first introduces the students into the user interface of the MPS tool. They need to know where to look for what type of information and how to trigger the basic actions of the IDE. Second, the logical structure of an MPS project is explained. The students learn the basic terminology, such as *node*, *model*, *module*, *language*, *generator*, and *solution*, their mutual relationships, as well as the way to set dependencies between them.

The first practical exercise exposes the students to the projectional editor. They need to learn the basics of manipulating projectional code (code completion, intentions, selection, etc.) while using a relatively simple language (like the Robot Kaja sample), as unfamiliarity with the editor would add complexity to the following, more involved, exercises. Essential debugging facilities, such as the reflective editor, the node explorer, and the hierarchy tree, are explained and used as well.

Language implementation itself is taught on a simple Robot Kaja language extension. The limitations of in-language abstractions are discussed and then a new command is created. The implementation starts with the STRUCTURE and the EDITOR. Then the GENERATOR is implemented to translate this new robot command into a piece of code in plain Robot Kaja language. The generator plan and transient models are explored along the way. A simple example of employing CONSTRAINTS and INTENTIONS follows.

The whole process of language creation is then repeated during the next exercise when a new language is built from scratch. Unless the client demands a customized domain to be used for the example, the students are guided through creating a state chart language that is generated into an XML format. This exercise adds the notions of parent–child relationships and references as well as the corresponding EDITOR and GENERATOR facilities related to them. Concept inheritance is also utilized.

As a wrap-up activity, the students get the task to implement a language from scratch by themselves. This quickly discovers the areas where the students lack the necessary skills and allows the teacher to fix this.

Apart from the possibility to choose a domain for the major exercise, the Advanced course does not allow for customization. This seems not to be of any limitation, since the fundamentals of the technology must be covered in their entirety.

Advanced Course The Advanced course explores some of the topics further and adds a few new ones, as outlined in Table 2. The goal of this course is to make the students competent in language implementation. The Shapes sample language is typically used as a starting point for the students to build nontrivial improvements in. The language should be enhanced with variable declarations and references, which leads to key principles like mapping labels, type system rules, scopes, and name uniqueness constraints. The code should be made more convenient to edit, which leads the students to using transformation menus, substitute menus, action maps, and node factories. Checking rules, quick fixes, intentions, and generator template modularization are also covered in this exercise.

The second part of the course consists of several independent chapters, each dedicated to a particular topic and explained using one of the MPS sample projects. The chapter on language testing, for example, explains how to create node, editor tests, and generator tests and lets the students practice it using the Robot Kaja

Table 2 Outline of the Advanced MPS course, which is a follow-up to the Introductory course and is based on one major example project and several single-purpose projects. The duration of the course is 16 h

Making the editor fluent with substitutions and transformations
Checking rules and quick fixes
Enhancing the structure with attributes
Commenting out code and documentation comments
Advanced structure: specialized links, customized presentation
Leveraging the full power of the SModel language
Concept functions
Dependency analysis
Testing and debugging of languages
The build language
Building MPS and IDEA plugins
Language versioning and migrations

language. The chapter on the build language uses the Calculator language sample to let the students create a language plugin as well as an independent IDE out of it. These chapters do not depend on one another and so can be included, excluded, or rearranged based on the participants' preferences.

The Process In general, the courses avoid spending excessive amounts of time explaining a single aspect of language definition, such as the EDITOR. Instead, they are organized into tasks. These tasks frequently require changes in several aspects of language definition, which ensures that the students iteratively revisit the individual aspects and gain additional details to the knowledge of the aspect that they have acquired in the previous iteration. This process is more engaging as it better balances the theoretical part with practical exercises.

It proved to be beneficial to adjust the exercises with the domain of the client. Not only does it increase motivation but also helps the students to feel more at ease with a tool that is very new to them. The feeling of actually prototyping a system that they will be implementing eventually for real engages the students and encourages concrete questions and discussions even beyond the scope of the exercise at hand.

Initially, the on-site courses took 3 days to complete, but the experience showed us that it is difficult to focus on a subject of this complexity for three subsequent days. Two days per course now seem to be the optimum, which naturally shifted some topics from the Introductory course into the Advanced one and left some topics out of the Advanced course completely.

Online courses tend to take about 25–50% more time to complete than their on-site variants. Despite that, we see a strong preference for online courses among the students.

Customized Courses Occasionally, JetBrains receives requests for highly customized training courses that would cover topics specified by the client and focus on the problem domain of the client. Sometimes this also includes consultancy around the architecture or a specific implementation of a DSL-based solution. These customized courses typically require expertise across several domains of the MPS technology stack, which is difficult to cover by a single lecturer, if the depth of the information given should stay the same in all the areas of the client interest. In order to offer the client the possibility to have each topic covered by an MPS expert on that particular subject, these customized courses are typically organized at the JetBrains premises.

Getting Feedback Collecting feedback from the students proved to be an essential way to improve the quality of the courses. The primary way to collect feedback from the online courses are the comments. These are conveniently located at every step of the material, which makes the context of the comment easy to refer to. Arguably, the feedback obtained immediately as the students go through the course is more accurate than the feedback collected at the end of the course. Collecting feedback during live on-site courses is the sole responsibility of the lecturer. She has to keep notes of the frequent questions and the issues that the students run into and incorporate them into the course materials. The usefulness of the after-course feedback forms that students fill out after they have finished the courses has been decreasing over time as the courses matured. They helped to direct and focus the courses at a coarse-grained level when at the beginning, but the answers to the feedback forms had become predictable as the contents of the course settled, and we have thus abandoned these feedback forms recently.

2 Teaching MPS in an Academic Environment

The first author has given a series of courses on the implementation of domain-specific languages at the University of Turku (Finland), Åbo Akademi University (Finland), the University of Bergen (Norway), and Queen's University (Kingston, Ontario, Canada) to groups of students both with and without a background on model-driven software development and software language engineering. A substantial part of these courses was devoted to language implementation with MPS.

2.1 A Course on Domain-Specific Languages

Position in Curriculum The course "Introduction to Domain-Specific Programming Languages" aims at attracting students to the field of software language engineering and exposing them to a wide range of techniques to implement languages in practice. A motivation to study language design and implementation is drawn from domain-specific languages (DSL), and a part of the course is devoted to DSL design principles.

The course has been lectured as both a short course and a full course, thus giving 3–5 ECTS study points. The course is optional and available to (computer science) students at all levels; several students majoring in economics and information systems have also attended the course. The only requirement to join the course is a background in any object-oriented programming language, though previous knowledge about programming paradigms, formal language theory, or theory of computation is beneficial.

Course Format A full version of the course, with 14 in-class sessions in total, is centered around 5 lectures and 9 hands-on tutorials (see Table 3). There are two 90-minute sessions per week, and 6 h per week are allocated for self-studying. Similarly to other courses related to software language engineering [2], the lectures are a combination of theory and practice, with theory explained on the slides and in discussions and then illustrated by the lecturer by live coding the examples [2]. Live coding is partly interactive, and suggestions from the students can be implemented. Having separate sessions in a computer class, where students could experiment on their own (rather than copying teacher's solutions) could be a better option [2]. Still, tutorial-like lectures do require students to use computers themselves to

Table 3 Outline of the university course

	Topic	Content	Book	Format
1	Zoo of DSLs	Examples, semantic models	[16]	Lecture
2	External DSLs	AST, IDE services	[16, 37]	Lecture
3	Eclipse Xtext	Grammars, EMF, validation, formatting	[7]	Tutorial
4	Eclipse Xtext	Type system, code generation	[7]	Tutorial
5	Spoofax	Typing rules for expressions		Tutorial
6	DSL design concerns	Best and worst practices	[22, 37]	Lecture
7	Metaprogramming	Rascal		Tutorial
8	Projectional editing	Examples, equation editor analogy		Lecture
9	MPS	Structure, editor	[10]	Tutorial
10	MPS	Type system	[10]	Tutorial
11	Code generation	XML, XSLT, MPS TEXTGEN aspect	[10]	Tutorial
12	MPS	GENERATOR aspect	[10, 37]	Tutorial
13	MPS	Extending Java		Tutorial
14	LOP in practice	Language-oriented programming	[37]	Lecture

implement explained techniques and mini-tasks given to students to experiment with implementation.

A short version of the course is structured in a flexible way and is adapted to the interests of students. The course was given twice at Bergen Language Design Laboratory at the University of Bergen in Norway: the first version had a hands-on tutorial on Eclipse Xtext and a demo of MPS, while the second iteration of the course focused on using MPS for extending Java. Students of the School of Computing of Queen's University in Kingston in Canada took a short version of the course with a hands-on tutorial on MPS.

Materials The course is based mainly on five books authored by industry experts, thus exposing students to a practitioner's view on language implementation: M. Fowler's book on domain-specific languages [16] is used for showcasing the theory of internal and external DSLs, the book on DSL engineering by M. Voelter et al. [37] gives material on external DSLs and IDE services, L. Bettini's book [7] is a foundation for tutorial lectures on Eclipse Xtext and Xtend, F. Campagne's book [10] is used to compile the tutorial on MPS, and S. Kelly and J.P. Tolvanen's book [22] is used in discussions on language design practices.

Slides of the course are available online at `dsl-course.org`.

Similar Courses There are multiple courses on domain-specific languages, software language engineering, and model-driven engineering given at various universities (see, e.g., [1], [2], [9], [12]). To the best of our knowledge, our course covers one of the widest ranges of language implementation techniques.

2.2 Teaching MPS Within the Big Picture of Language Engineering

The course is divided into four principal parts. The *first part* is an extensive zoo of domain-specific languages including formatting languages, graphical rendering languages, data description languages, and business process languages. A wide coverage of domains helps students get an informal idea about what a DSL is. A pseudo-realistic example [4] that explains the need, adoption, and evolution of domain-specific languages and tools is presented.

The *second part* of the course focuses on general properties of domain-specific languages. It starts with a discussion on internal and external DSLs, advantages of using them and possible problems, and design guidelines [20, 21] and continues with an overview of integrated development environments (IDEs). Students are familiarized with technical requirements for standard IDE services such as syntax-aware editing, code formatting and folding, code completion, code navigation and hyperlinking, code outlining, name refactoring, and automatic code corrections. An informal comparison of how these features can be implemented in several language workbenches follows later in the course.

The second part of the course also introduces a simple external language—*Entities Language* [7]—that will be later implemented using several language engineering tools. This choice of (a trivial) Entities Language is dictated by a learning objective of demonstrating different aspects of language implementation (such as the definition of a type system and code generation of object-oriented code) rather than that of exercising language design skills. A code example in Entities Language is given below:

```
entity Person {
   text name
   number YoB
   name = "John"
   YoB = 1900
}
entity Employee extends Person {
   number salary
   name = "Mary" // inherited field
   salary = 30 * (20 + 80)
   print salary
}
```

Each `entity` corresponds to a class in object-oriented programming, and an `entity` can extend another declared `entity`. The language has three primitive data types—`number`, `toggle`, and `text`—that correspond respectively to the integer, Boolean, and string types in common programming languages. Moreover, each `entity` becomes a data type and fields of that type can be declared. The body of an `entity` is a sequence of fields declarations and statements (assignment and

Table 4 Topics covered in the course when discussing different language workbenches

Tool	Entities definition	Expressions sublanguage	Code generation	Typing rules	Scoping rules
Eclipse Xtext	Yes	No (constant expressions only)	Yes (M→T)	Yes[a]	Yes[a]
Spoofax	Yes	Yes	No	Yes[b] (expressions only)	No
Rascal MPL	Yes	Yes	Yes (evaluation[c])	No	No
MPS	Yes	Yes[d]	Yes (M→T, M→M)	Yes	Yes

[a]imperative Java/Xtend code [7]
[b]declarative NaBL2 code [24]
[c]evaluation defined for both abstract and concrete syntaxes
[d]imported from MPS BaseLanguage
M→T, model-to-text transformation; M→M, model-to-model transformation

`print`), which can occur in any order. From within the body of an `entity` that extends another `entity`, it is possible to access the fields declared in that other `entity`, following the standard semantics of class inheritance. Entities Language is supposed to be transpiled into an object-oriented language, for example, Java.

The *third part* of the course discusses language workbenches Eclipse Xtext [15] and Spoofax [19, 41] and gives an overview of metaprogramming language Rascal [23]. Our hypothesis is that the students will better perceive techniques of language engineering in MPS when they already are familiarized with language implementation approaches in a text-based language workbench. Instead of choosing a single language workbench for this purpose, a combination of three tools is showcased to the students, with different focus when discussing each of them (see Table 4 for summary).

Eclipse Xtext Implementing Entities Language in Eclipse Xtext starts with a grammar definition. To keep the focus on language implementation issues (such as IDE services) rather than on grammar rules for arithmetical expressions with priority of operations,[1] expressions can only be constants. A point is made that grammar rules correspond to EMF Ecore [17, 33] object model, thus preparing the students to the object-oriented way on defining concepts in MPS.

Type checking and scoping rules are implemented using imperative code in Java and Xtend [8]. Code formatting is implemented in Xtend using an internal DSL that ships with Xtext. This is expected to bring students' attention to the *language-oriented programming* approach used for language definition: this point is again

[1]This is not completely trivial because Xtext is based on a recursive descent parser that forbids left recursion in grammar rules; cf. syntactic rules in Spoofax mentioned later.

mentioned later in the course when explaining how language aspects are defined in MPS using DSLs tailored for each aspect [13]. Entities Language is transpiled in a straightforward way into Java. Implementation of code generation uses multiline template expressions of Xtend [7, 17] and is thus a model-to-text transformation.

Spoofax Students are then introduced to a more formal way of defining typing rules for expressions when language workbench Spoofax is discussed.[2] The grammar of the Entities Language is redefined in Syntax Definition Formalism SDF3 [11, 19] that has built-in capabilities for defining priority of arithmetical operations. Since Spoofax uses Generalized LR parsing algorithm [18], context-free rules of arbitrary form can be used. This allows defining rules for expressions in a natural way (e.g., $Expr \to Expr + Expr$). Typing rules are defined using a declarative language NaBL2 [24] that enables definitions similar to a mathematical notation (along the lines of $e_1 : \text{INT},\ e_2 : \text{INT} \Rightarrow e_1 + e_2 : \text{INT}$). An informal comparison with the type system implementation in Xtext is made at this point, and students are expected to be able to value the benefits of declarative specifications, where the gap between the formalization and the implementation of a type system is reduced [8, Sect. 7.3]. This is yet another preparatory step for the students before they are familiarized with MPS.

Despite a powerful model transformation mechanism of Spoofax, code generation techniques are not discussed at this point. Doing otherwise would require explaining model transformations twice: both for Spoofax and then later for the case of MPS.[3]

Rascal MPL The course continues with an implementation of Entities Language using metaprogramming language Rascal, with the focus on yet another aspect of expressions sublanguage—that of their evaluation. While discussion of expressions evaluation was possible already in the case of Xtext, this topic is left for Rascal to demonstrate the idea of code quotations: a method with signature `eval((Expr)'<Expr e1> + <Expr e2>')` quotes the concrete syntax of an addition expression and can perform calculations on variables `e1` and `e2`. This

[2]In an alternative curriculum, we could have only showcased a single text-based language workbench in the course—Eclipse Xtext—with type system and scoping rules defined using Xsemantics [8] and expressions sublanguage imported from xBase [14]. A decision has been made, however, to expose students to a larger variety of tools.

[3]Though code generation mechanisms in both Spoofax and MPS are similar *conceptually*, there are two important differences. First, in MPS, code to be generated is essentially a quotation, and only valid fragments of code can be emitted, whereas Spoofax allows emitting any strings [28]. Second, in MPS, code to be generated is a *sample output*, whose varying parts are explicitly marked with generation *macros*, the machinery of which is defined in the Inspector window. For example, for the STRUCTURE definition `concept VarDecl{type, name}`, a possible GENERATOR definition is `private $MACRO[int] $MACRO[amount];`. In Spoofax, the output is essentially an interpolated string: `VarDecl(type, name) -> private [type] [name];`. Hence, to avoid possibly confusing the students, a decision has been made to explain code generation only for the case of MPS. In a more extensive course, however, explaining both mechanisms seems to be beneficial.

"light" exposure to the notion of quotation is again preparing the students for the language implementation in MPS.

By this point in the course, students have an understanding of what language implementation comprises of and are expected to be somewhat fluent in the language engineering terminology. The course can then continue with the *fourth part*, which is devoted to the language implementation in MPS.

Explaining Projectional Editing Before starting to implement Entities Language in MPS, a detailed discussion on projectional editing takes place. We find this important since the projectional editor of MPS both is its most distinctive and powerful feature and the one that prevents a wide adoption of MPS due to its perceived clumsiness [29].

The discussion on the projectional editing starts with examples of non-programming environments, which include:

- sound editing software (e.g., Adobe Audition[4]) that supports the graphical editing of sound waves (thus, one projection is audible and the other one is visual);
- rich text editors with support of textual, graphical, and tabular notations;
- file system explorers that have several visual representations of file system elements (e.g., "large icons," "list") and allow for graphical editing of the file system (by dragging and dropping the files and folders onto each other);
- XML-based file formats (e.g., Microsoft PowerPoint), where an animation can be edited both in the tool's visual interface and in code;[5]
- vector graphics editing software (e.g., Inkscape[6]) that allows both the visual editing of an image and a direct manipulation of its SVG representation;
- diagrams editing software (e.g., SmartArt tool of Microsoft Word) that enables the visual and textual editing of the diagram's structure and elements;
- HTML editing tools (in the style of Microsoft FrontPage) that allow both the rich text-based and the code-based editing of HTML code.

The following examples of programming environments that use projectional editing are mentioned in the course:

- office database management systems (e.g., Microsoft Access), where table definition queries are themselves represented as tables, and SQL queries have textual, tabular, and graphical projections;
- computer algebra systems (e.g., PTC Mathcad[7]) that allow visually enhanced control flow instructions with a graphical rendering of mathematical formulae and support diagram notations (such as plots) directly in code;

[4]http://adobe.com/products/audition.

[5]https://docs.microsoft.com/en-us/office/open-xml/working-with-animation.

[6]https://inkscape.org.

[7]https://www.mathcad.com.

- rapid application development studios and form builders that allow both visual and text-based editing of GUI forms;
- Scratch-like programming environments [27], where the code is constructed from graphical blocks corresponding to the programming language statements.

Analogy with Equation Editor A more detailed explanation of projectional editing is done using the analogy of Equation Editor in Microsoft Office. To explain how a formula

$$\sum_{\square}^{\square} \square$$

with three placeholders can be entered by a user of an Equation Editor, the following object model is suggested: a class `Sum` with three fields `lowerBound`, `upperBound`, and `expr`, each of type `Expression`. The graphical rendering of this formula is then a *projection* of an instance of this class. A point here is made that the definition of class `Sum` repeats the structure of a concept for a summation expression that could be defined in MPS.

Continuing with the object-oriented metaphor, the following example of a matrix rendering in the Equation Editor is considered:

$$\begin{bmatrix} \square/\square & \sqrt{\square} \\ \square & \square^{\square} \end{bmatrix} \tag{1}$$

This matrix can be considered as an edited projection of a two-by-two matrix with four "basic" placeholders

$$\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix} \tag{2}$$

which is represented as an instance of class `Matrix2x2` with four fields `pos11`, `pos12`, `pos21`, and `pos22` of type `Expression`. In the Equation Editor, editing of projection (2) to get the desired projection (1) is done either by using menu commands and palette or by typing a trigger for a specific kind of placeholder. Typing those triggers ("`/`," "`sqrt`," "`^`") instantiates, respectively, classes `Division`, `Radical`, and `Exponentiation` that all extend class `Expression`. The (now updated) object model of (1) has field `pos11` of type `Division`, field `pos12` of type `Radical`, and field `pos22` of type `Exponentiation`. This example is used as an intuitive introduction to concept inheritance in MPS.

Another example anticipates discussion on aliases of concepts in MPS. Projection of class `SineFunction` with field `expr` of type `Expression` can be entered in the Equation Editor by typing symbols `s`, `i`, and `n`. The editor treats this combination of letters as a trigger (or alias) of concept `SineFunction`, which is then instantiated and projected as "`sin` $\square$."

To anticipate left- and right-side transformations that can be defined for an editor of a language in MPS, yet another example of the Equation Editor is considered. When an integer expression `2` is already entered by the user in the Equation Editor, typing caret symbol "^" will force the editor to transform the projection of integer constant into a projection $2^{\square}$ of exponentiation expression with one placeholder.

Finally, projectional editor of MPS is explained by giving an example of an imperative programming language construct. An instance of this construct, say, *if statement*, is represented as a table where a cell is assigned to every lexical element (token) present in the example. Three kinds of cells are then distinguished: immutable cells that represent keywords and special symbols, indentation cells, and mutable cells that contain elements of the abstract syntax of the construct.

These examples are expected to give enough background to students so that they can use a projectional editor of MPS or another tool that features a projectional editor.

Entities Language in MPS At this point, Entities Language can be implemented in MPS. Following the analogy of the Equation Editor, each construct of Entities Language is considered as a class in the object model induced by the language. A point is made about resemblance to Xtext, where grammar rules become classes (instances of EMF Ecore `EClass`) and features of rules become fields of those classes. It is important to convey to students that while in Xtext the object model is populated during the parsing, resulting in an abstract syntax tree that can be further manipulated, in MPS the user edits the abstract syntax tree directly. Another point worth making about comparing language implementation in Xtext and in MPS is that Xtext is mainly based on a general-purpose imperative language to implement language aspects, while MPS follows the approach of language-oriented programming [13].

Expressions Unlike implementations of expression sublanguage in other language workbenches studied in the course, we use language composition [37] to import expressions from MPS BaseLanguage. This allows students to focus on what is distinctive for language definition in MPS rather than to meticulously define the inductive structure of expressions. Nonetheless, an interested student can be given a task to implement expressions sublanguage from scratch in order to practice defining editor actions to achieve text-based-like entry of expressions and then to reimplement the editor using grammar cells [39].

Language Aspects While defining the STRUCTURE and EDITORs for concepts of Entities Language is rather straightforward, lecturer's attention should be given to checking whether the students correctly type in code snippets in MPS projectional editor. A home assignment, similar to some of the exercises discussed in [6], can be given to students to practice their typing experience with MPS. To showcase rich rendering capabilities of the projectional editor, in addition to stylesheets and conditional cells, Java Swing components are used; for example, an editor for a Boolean value features a button component that switches the value. Implementing an action listener for this button demonstrates a less trivial use of the *Inspector* window. To nurture a more productive typing of code in MPS, students are asked to use extensively *Intentions* menu when defining STRUCTURE and EDITORs for concepts.

To reiterate the object-oriented nature of concepts, several methods in the BEHAVIOUR aspect are defined. This is followed by creating CONSTRAINTS and TYPESYSTEM rules for concepts. In addition to implementing standard rules for typing expressions, Entities Language is modified to showcase how physical units (e.g., `number[kg] x; x = 10 CHF`) can be type-checked, with the goal of practicing the SModel language of MPS. A particular attention is given to quotations and antiquotations, which have been already briefly introduced when discussing metaprogramming language Rascal earlier in the course.

Explaining Code Generation Code generation is perceived as one of the most involved parts in the language definition process. Trying to provide a bridge to already perceived ideas of model-to-text transformations with Xtext, the course first explores TEXTGEN aspect of MPS, where the Entities Language is transpiled into Java. A point is made on why generating textual output may be inferior to using GENERATOR aspect, where target language code is guaranteed to be syntactically correct.

To explain model-to-model transformations in MPS, a dive into XML and XSL transformations is made. Students are expected to practice using XSLT for generating both XML documents and textual output (e.g., Java code). An advanced exercise here would be to ask students to design their own XSLT-like transformation language and then to write an XSL transformation that will convert student's custom language into XSLT. Table 5 can be used to initiate discussion on the resemblance between different XML artifacts, language aspects in MPS, and notions of application software. This brings an alternative way of teaching MPS (discussed in Sect. 2.3) that starts with treating languages defined in MPS as user interface specifications [5].

Even after practicing XSL transformations, we find it useful to explain GENERATOR aspect anew on a blackboard without using MPS. We give examples of target language code and show how we could "annotate" it to form an intuition for macros and template fragments.

To let students practice defining language constructs and generators for them "in the wild" and to reiterate the importance of language composition, the course contin-

Table 5 Conceptual resemblance of XML artifacts, aspects of MPS, and application software

XML artifact	MPS aspect	Application software
XML document	STRUCTURE	Data model
XML schema or DTD	TYPESYSTEM, CONSTRAINTS	Data validation
CSS stylesheet	EDITOR	Graphical user interface
XSL transformation (XML to text)	TEXTGEN (M→T)	Data serialization
XSL transformation (XML to XML)	GENERATOR (M→M)	Business logic
Document Object Model	Abstract syntax tree	Data, metadata

ues with extending MPS BaseLanguage. We offer as assignments adding physical units, `nameof` operator [25], unconventionally layout imperative statements, and other constructs to BaseLanguage.

2.3 *A Different Curriculum: Teaching MPS as a Programming Tool En Soi*

We describe below an outline of an alternative version of the course, where introducing MPS does not require previous knowledge on other language workbenches.

Reflecting the typical patterns for building software languages [35], the course is divided into three parts:

- MPS as a tool for textual user interfaces [5];
- MPS as a modelling tool [34];
- MPS as a tool for extending existing languages [31].

This choice and order of the parts is dictated by a desire to avoid starting the course with language engineering terminology. In the first part of the course, the students' task is to create a textual user interface for an invoice calculator in MPS. Discussing language aspects at this point avoids mentioning projectional editing [29, 38]. ACTIONS aspect is not mentioned until the second part of the course, where it comes naturally in one of the examples. INTENTIONS aspect is discussed, and a point about importance of domain-specific error support [16, 40] is made. Implementation of the invoice calculator in Microsoft Excel could be discussed for comparison.

Code Generation Code generation is covered in several parts of the course. Discussion starts with the TEXTGEN aspect, where an RTF generator for the invoice calculator is defined. It continues with the GENERATOR aspect, where generators for the invoice calculator to XML and Java are given. Java code generation deliberately comes after RTF and XML to address a frequent opinion of the newcomers that MPS only allows generating Java code.

Projectional Editing Projectional editing often alienates beginners by its perceived clumsiness and limitations [29]; that is why it is not formally introduced until this moment of the course. A discussion *pro* projectional editing follows, stating that most of a traditional editor functionality is replicated in MPS [36].

Entities Language In the second part of the course, MPS is used as a modelling tool to implement domain-specific languages. A running example is Entities language; code generators into Java, JavaScript, and Visual Basic for Applications are discussed. This selection of target languages is motivated by the need to communicate to students that MPS can be used to generate web applications as well as legacy code. A metaphor of object-oriented programming counterparts in MPS is revised, and the students are encouraged to focus on notions used in language engineering [26]. Code generation with the GENERATOR aspect is revisited, with discussion on reductions, quotations, and mapping labels. The discussion continues with the TypeSystem and the DataFlow analysis for the Entities Language. This part is concluded by discussing language Migration in MPS.

In the third part of the course, MPS is used to extend Java [31] and Entities Language, where Actions, Constraints, and Generator aspects are discussed anew.

3 Lessons Learned

We present below a summary of lessons learned based on our experience in teaching language engineering with MPS in industry and academia:

L1. *Low ceremony free courses* attract the participants who only want to test the waters before fully committing to learning the technology and who would turn the technology down without trying otherwise.

L2. *Going online* has increased the reach of the courses by an order of magnitude.

L3. Paying attention to *step-by-step guidance* is crucial even at later stages, since the participants tend to skip unclear steps or forget things, if not repeated several times in different contexts.

L4. *Teaching MPS in a wide context of other language engineering tools*, such as language workbenches Eclipse Xtext, Spoofax, and Rascal, gives students a fuller picture of this area. Most importantly, by the time when MPS is introduced in the course, students have already seen what language implementation is comprised of and acquired some language engineering terminology. This enables concentrating on implementation techniques that are distinctive in MPS.

L5. *Explaining projectional editing by discussing analogies* among both non-programming and programming-related environments seems to facilitate perception of MPS projectional editor. An analogy with an equation editor in a word processor enables introducing object-oriented view on language concepts and the notions of concept aliases and side transformations.

L6. *Explaining code generation with XML-based examples and XSL transformations* allows the students to explore model-to-model transformations. Availability of (online) tools to run XSL transformations, textual representation of XML trees, and powerful mechanisms of XSLT facilitate students' experience.

L7. *Using MPS to extend an existing general-purpose language* (e.g., MPS BaseLanguage) seems to interest students as they can see an immediate quasi-practical use of the acquired language engineering skills.

L8. Covering at least the basics of *domain engineering* motivates the participants to learn the practicalities of the concrete tool. It will also help them imagine how the principles and tooling could be applied to their projects and their infrastructure.

L9. *Language design concerns* should be explored as part of the initial discussions over nontrivial example projects. The pros and cons of the alternative approaches should be presented as well as their consequences to other parts of the project.

4 Concluding Remarks

We presented our experiences in teaching MPS in two inherently different environments. Courses and trainings given in an industrial setting are aimed at experienced developers and business experts, are often adjusted for a particular business domain, and are designed around acquiring practical language implementation skills by the participants. The academic setting tends to give a broader yet less detailed overview of several language implementation techniques, with the goal that students grasp fundamental concepts and differences between approaches and tools.

Attracting students to taking any form of training in the MPS technology has constantly been the biggest challenge. Numerous beginner-level questions on the MPS discussion forum indicate that a large number of professionals attempt to climb the steep learning curve on their own, which frequently leads to suboptimal solutions, shallow opinions, and a lot of frustration.

It remains to see whether joint efforts from both industry and academia in teaching MPS would benefit all the stakeholders. Perhaps a first step already taken in this direction is the availability of teaching materials [3, 30] online.

References

1. Acher, M.: Domain-Specific Languages, course materials. Available at: http://mathieuacher.com/teaching/MDE/201516/DSLAndXtext.pdf
2. Bagge, A.H., Lämmel, R., Zaytsev, V.: Reflections on courses for software language engineering. In: MODELS Educators Symposium 2014, pp. 54–63 (2014)
3. Barash, M.: Introductory course on domain-specific programming languages, course materials. 2017–2020. Available at: http://dsl-course.org
4. Barash, M.: A tale about domain-specific languages, blog post (2018). Available at: https://medium.com/@mikhail.barash.mikbar/a-tale-about-domain-specific-languages-bde2ace22f6c
5. Benson, V.M., Campagne, F.: Language workbench user interfaces for data analysis, PeerJ 3:e800 (2015)
6. Berger, T., Voelter, M., Jensen, H.P., Dangprasert, T., Siegmund, J.: Efficiency of projectional editing: a controlled experiment. SIGSOFT FSE 2016, pp. 763–774
7. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing (2016)
8. Bettini, L.: Type errors for the IDE with Xtext and Xsemantics. Open Comput. Sci. **9**(1), 52–79 (2019)
9. Cabot, J., Tisi, M.: The MDE diploma: first international postgraduate specialization in model-driven software engineering. CS Education 2011

10. Campagne, F.: The MPS Language Workbench, Vol. 1. CreateSpace Independent Publishing Platform (2014)
11. de Souza Amorim, L.E., Visser, E.: Multi-purpose Syntax Definition with SDF3. SEFM 2020, pp. 1–23
12. Dingel, J.: *Beyond Code: An Introduction to Model-Driven Software Development*, course materials. Available at: http://research.cs.queensu.ca/home/dingel/cisc836_W20/index.html
13. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm (2004). Available at: https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf
14. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java, GPCE 2012, pp. 112–121
15. Eysholdt, M., Rupprecht, J.: Migrating a large modeling environment from XML/UML to Xtext/GMF. SPLASH/OOPSLA Companion 2010, pp. 97–104
16. Fowler, M.: Domain-Specific Languages. Addison-Wesley, Boston (2010)
17. Gronback, R.C.: Eclipse Modeling Project – A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2009)
18. Grune, D., Jacobs, C.J.H.: Parsing Techniques — A Practical Guide, pp. 1–643. Springer, New York (2008)
19. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs, OOPSLA 2010, pp. 444–463
20. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (2009)
21. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Softw. **26**(4), 22–29 (2009)
22. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
23. Klint, P., van der Storm, T., Vinju, J.J.: Rascal: a domain specific language for source code analysis and manipulation. In: SCAM 2009, pp. 168–177
24. Konat, G., Kats, L.C.L., Wachsmuth, G., Visser, E.: Declarative Name Binding and Scope Rules SLE 2012, pp. 311–331
25. Kulikov, P., Wagner, B., De George, A., Wenzel, M.: nameof expression. C# Programming Language Reference. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof
26. Lämmel, R.: Software Languages: Syntax, Semantics, and Metaprogramming. Springer, New York (2018)
27. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Trans. Comput. Educ. **10**(4), 16:1–16:15 (2010)
28. Metaborg, Concrete syntax in stratego transformations. Available at: http://www.metaborg.org/en/latest/source/langdev/meta/lang/stratego/concrete-syntax.html
29. Minör, S.: Interacting with structure-oriented editors. Int. J. Man Mach. Stud. **37**(4), 399–418 (1992)
30. Pech, V.: JetBrains MPS Elementary Course, online course. Available at: https://stepik.org/course/37360/
31. Pech, V., Shatalin, A., Voelter, M.: JetBrains MPS as a Tool for Extending Java, PPPJ 2013, pp. 165–168
32. Ratiu, D., Pech, V., Dummann, K.: Experiences with teaching MPS in industry: towards bringing domain-specific languages closer to practitioners. In: MODELS 2017, pp. 83–92
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF – Eclipse Modeling Framework. Addison-Wesley, Hoboken (2008)
34. Voelter, M.: Fusing Modeling and Programming into Language-Oriented Programming – Our Experiences with MPS, ISoLA 1, pp. 309–339 (2018)

35. Voelter, M.: High-Level Structure of DSLs: Three Patterns (2017). Available at: https://languageengineering.io/high-level-structure-of-dsls-three-patterns-7375c8baa2d3
36. Voelter, M., Lisson, S.: Supporting Diverse Notations in MPS' Projectional Editor. GEMOC@MoDELS 2014, pp. 7–16
37. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages (2013)
38. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards User-Friendly Projectional Editors. SLE 2014, pp. 41–61
39. Voelter, M., Szabó, T., Lisson, S., Kolb, B., Erdweg, S., Berger, Th.: Efficient development of consistent projectional editors using grammar cells. SLE 2016, pp. 28–40
40. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., van Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with MPS. Softw. Syst. Model. **18**(1), 585–630 (2019)
41. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)