Programming
vs.
That Thing Subject Matter Experts Do

Markus Voelter

independent/itemis, Oetztaler Strasse 38, 70327 Stuttgart, Germany
voelter@acm.org

Abstract. Keywords: Domain Specific Language · End-user Programming · Language Engineering.

1 The role of subject matter experts

Subject matter experts, or SMEs, own the knowledge and expertise that is the backbone of software and the foundation of digitalization. But too often this rich expertise is not captured in a structured way and gets lost when translating it for software developers (SDs) for implementation. With the rate of change increasing, time-to-market shortening and product variability blooming, this indirect approach of putting knowledge into software is increasingly untenable. It causes delays, quality problems and frustration for everybody involved. A better approach is to empower *subject matter experts* to capture, understand, reason about data, structures, rules, behaviors and other forms of knowledge and expertise in a precise and unambiguous form by providing them with tailored software languages (DSLs) and tools that allow them to directly edit, validate, simulate and test that knowledge. The models created this way are then automatically transformed into program code. The *software engineers* focus their activities on building these languages, tools and transformations, plus robust execution platforms for the generated code. Fig. 1 shows the overall process.

TODO: Refer to Manifesto when it is done

2 (Where) Does this work?

Building the necessary tooling and automation requires investment, and this investment must pay off for it to make economical sense. This is why this approach only works in domains that have the following characteristics. First, the subject matter that goes into the software has to have a minimum **size** and **complicatedness**. Often this means that there are people in the company who consider themselves the **experts** in that subject matter. It is they who everybody asks about details in the domain. The second criterion is that this subject matter as a whole will remain **relevant over time** and that the business intends to

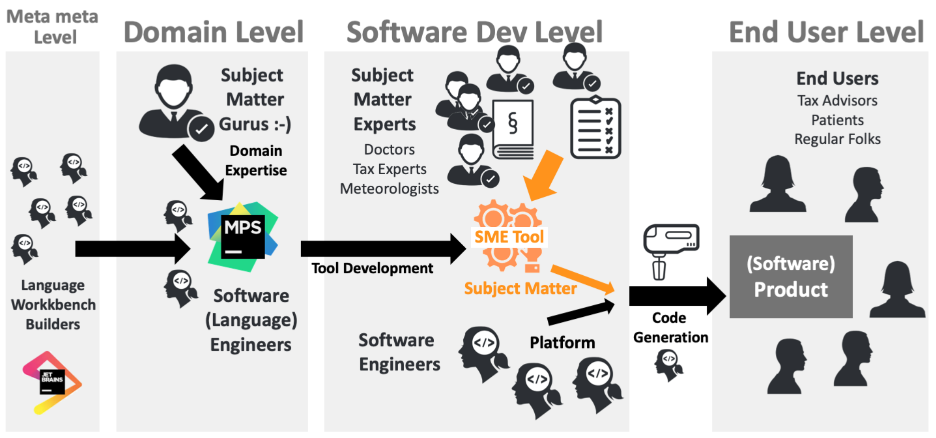


Fig. 1. The process from the SME's brain into software, based on tools and platforms developed by software engineers.

continue developing software in that domain for a reasonably long time. Finally, even though the subject matter as a whole is relevant for a long time, a degree of **evolution** or **variety** within the domain is usually needed for the approach to make sense. We have seen this approach used in the following domains, among others:

In insurances, DSLs are used by insurance product definition staff to develop a variety of ever evolving insurance products [4]. With increased differentiation and tailoring of products, these become more and more complicated while at the same time increasing in variety and number. The company itself is in the insurance business for the long run. [Reference to detailed case study coming once MPS book is published**TODO:**]

In healthcare, DSLs are used by doctors and other healthcare professionals to develop treatment and diagnostics algorithms inside digital therapeutics apps. The company intends to grow over years and develop a large number of these algorithms and apps. The subject matter is large and complicated because it captures medical expertise. Details about the French-American company Voluntis see [8].

In public adminstration, government agencies are certainly in it for the long run, while legislation for public benfits and tax calculation changes and evolves regularly. The agencies are full of experts who disambiguates and formalizes the law that is written by the government and its interpretations by courts using DSLs [1]. Similarly, the service providers who develop software for tax advisors has the same challenges and uses DSLs as well. [Reference to detailed case study coming once MPS book is published**TODO:**]

In payroll, the applicable regulations that govern the deducations from an employee's gross salary and the additional taxes and fees they have to pay are just as complicated, long-lasting and ever changing as tax law. Service provides who develop payroll software therefore also employ whole departments full of experts and benefit from DSLs. [Reference to detailed case study coming once MPS book is published**TODO:**]

For an overview over this approach and a couple of easily readable case studies, see my InfoQ article [7].

3 Typical DSL Architecture

Many of the DSL I have built follow the general approach that is outlined in Fig. 2. The models created by the SMEs end up being a functional core of the application, either via generation or via interpretation. That core implements a (manually defined) API that is used by a driver component to invoke the DSL-derived code. The driver interacts with users, and other systems and is often also responsible for managing and persisting state. Indeed, many of our DSLs are funclarative [5] where small, simple calculations are done with a functional-style language (so users don't have to care about effects at this level) and when things get more complicated, we add declarative first-class concepts to the language.

In order to avoid reinventing the wheel with regards to the core functional expressions, we embed, and then extend a reusable language KernelF [9].

If the DSL cannot be scoped to handling only the functional parts, we usually rely on forms of state machines. Generally it is a good idea to rely on established programming paradigms and DSL-ify them instead of inventing new fundamental paradigms.

4 Can SMEs use DSLs?

This paper is not about justifying the approach from a technical or economical perspective. Instead I want to focus on whether the SMEs are able to change from their typically imprecise, non-formal approrach of specifying requirements using Word, Excel, User Stories or IBM Doors to this DSL-based approach.

In my experience, based on the experiences mentioned above, plus a few more, the answer to this question is a clear yes, at least for the majority of subject matter experts I have worked with. But a key question is: to what extent do the subject matter experts who use our DSLs and tools have to become programmers? Do they have the *skills* to be programmers (hint: most don't). Do they *want* to become programmers (hint: most don't). But we still expect them to use "languages" and IDE-like tools. So:

Which parts of what we call programming do they have to learn? How is SME'ing different from programming, and where does it overlap?

In the remainder of this paper I will try to provide some insight.

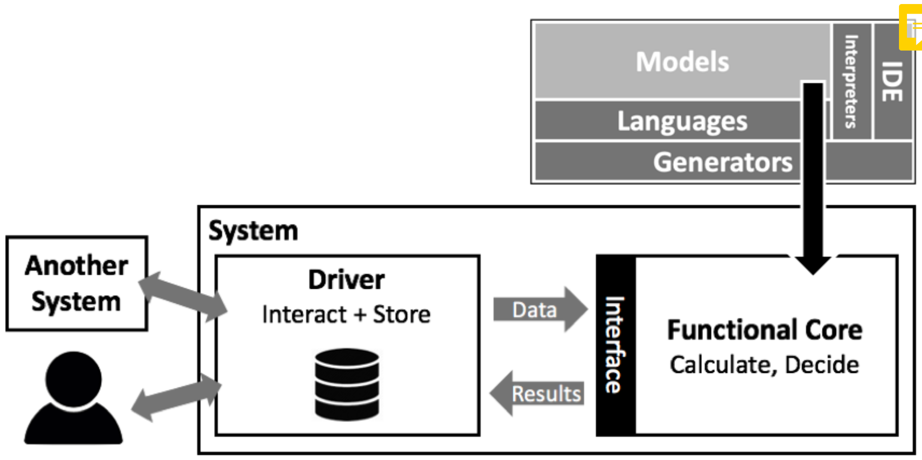


Fig. 2. Typical high-level systems architecture, where the models expressed with DSLs are transformed into code that forms the core of a larger application.

5 Difference between programming and SME'ing

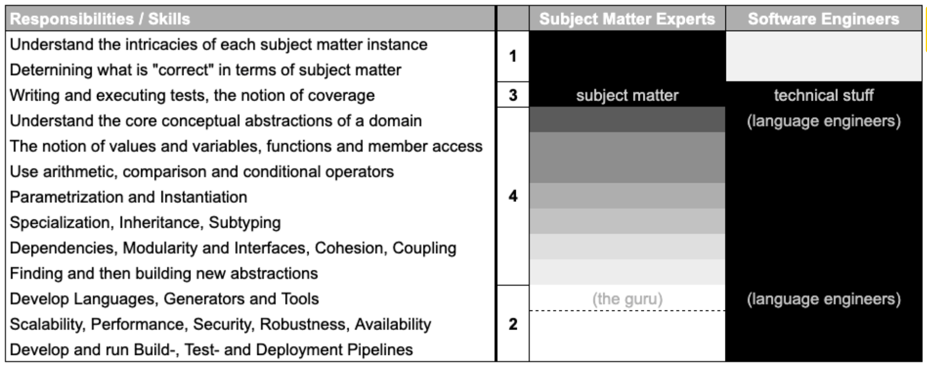
5.1 Skills and Responsibilities

Fig. 3 shows the differences in responsibilities of DSs and SMEs. The darker the shade, the more responsibility the respective community has for that concern. Let's start our discussion with the two black and white cases, those where there is no overlap.

Region 1, SME only We start with region 1, which is completely the responsibility of SMEs. They have to understand every particular example, case, situation, and exception of the subject matter they are describing with the DSL. This is their natural responsibility, this is why they exist. SDs on the other side should not have to care at all. Achieving this separation – and then optimizing the tasks of both communities – is the reason for using DSLs and tools in the first place. Related to this, the SMEs are also in charge of determining what consistutes correct behavior in the subject matter. SMEs take responsibility for what should go into test cases and for their completeness.

Region 2, SE only Let's move on to region 2, which is completely the responsibility of SDs. Building and operating automated CI pipelines that build and package the software and run tests is nothing the SME should be concerned with, except maybe getting emails if tests fail (after they have run correctly in their local environment, otherwise they should never reach the CI server).

The same is true for taking care of performance, scalability, safety, security, robustness and availability, all the operational (aka non-functional) concerns of the final software system. Keeping the subject matter segregated from these technical aspects of software is a key benefit of the approach, and it is clear that



Responsibilities / Skills		Subject Matter Experts	Software Engineers
Understand the intricacies of each subject matter instance	1	■	░
Deternining what is "correct" in terms of subject matter	1	■	░
Writing and executing tests, the notion of coverage	3	subject matter	technical stuff
Understand the core conceptual abstractions of a domain	4	▓	(language engineers)
The notion of values and variables, functions and member access	4	▓	■
Use arithmetic, comparison and conditional operators	4	▓	■
Parametrization and Instantiation	4	▒	■
Specialization, Inheritance, Subtyping	4	▒	■
Dependencies, Modularity and Interfaces, Cohesion, Coupling	4	░	■
Finding and then building new abstractions	4	░	■
Develop Languages, Generators and Tools	2	(the guru)	(language engineers)
Scalability, Performance, Security, Robustness, Availability	2	□	■
Develop and run Build-, Test- and Deployment Pipelines	2	□	■

Fig. 3. Comparing programming (what software engineers do) from whatever not-yet-named activity subject matter experts should do to directly contribute to software. Darker shade means "is more relevant".

this should be handled in platforms, frameworks and code generators – all clearly the domain of SDs.

Finally, the development of the DSLs and tools that will then be used by the SMEs for capturing, analysing and experimenting with subject matter is the responsibility of SDs. It might not be the domain of *all* of the software engineers, but maybe that of a certain specialization called language engineers who specialize in developing languages, IDEs, interpreters and generators. Also, a very few of the subject matter experts – we sometimes refer to them as gurus – have to help survey, understand, analyse and abstract the domain so that the language engineers can build the languages. But the regular SME, the dozens or hundreds that many of our customers employ, are not involved with this task.

Region 3, Testing Let's now look at region 3, one that apparently has full shared responsibility. However, this is a bit misleading. Indeed, both communities have to understand the purpose of testing, what a test case is, and appreciate the notion of coverage, i.e., understanding when they have enough tests to be (reasonably) sure that there are no more (reasonably few) bugs in the logic. But of course the SMEs care about this *only* for subject matter (on DSL/model level) whereas the SDs care about it only in the platform, frameworks and generators. So they both have to understand the concept of testing, but there are no artifacts for which they have shared responsibility.

Region 4, shared skills We are now ready to look at the interesting part, region 4. Everything in this region is native to programming and software engineering. So SDs care. But they are also relevant, to different degrees, to SMES when they use DSLs. Let's explore them in detail.

Of course the SMEs have to understand the conceptual abstractions of the domain, because otherwise they cannot use the language. Understanding abstractions is not easy in general, but because in a DSL these abstractions are closely aligned with the subject matter, the SMEs – in my experience – are able to understand it. Maybe not every little detail (which is why the box is dark grey and not black) but sufficiently well to use the language. The SDs have to understand these abstractions as well, especially the language engieneers. Those who build the execution platform can probably deal with a black-box view of the generated code.

A note: the fact that there is shared understanding about the core abstractions of the subject matter in which the SMEs and the SDs co-create software is a major reason why DSLs are so useful here. The language definition, and its conceptual cousins, the core abstractions, are a great unambiguous and clearly-scoped foundation for productive collaboration between the two communities. So when I write "xyz is the responsibility of SME/SD", it doesn't mean that there is not a *joint overall responsibility* of both communities together to deliver useful (subject matter) and robust (-ilities) software.

Back to region 4. No real-world subject-matter focused DSL I have ever seen can make do without understanding the notion of values, and some notion of functions (entities that produce new values from inputs). It doesn't matther whether we're talking specifically about (textual) functions, (Excel-style) decision tables or (graphical) dataflow diagrams. The good thing is that essentially everybody has come across these at school or at university, so this is usually not a huge challenge.

Similarly, the understanding of dot expressions to mean `the member X of entity Y` or `do Z with entity Y` is very hard to avoid, because working with parts of things or performing activities on things is almost unavoidable. For many SMEs, this is harder to get used to. We've experimented with literally writing `<member> of <object>` instead of `<object>.<member>` but this is annoying because of scoping: with the latter syntax, you can easily scope the code completion menu to members of `object` because you write the object first, whereas in the former case the code completion for the member has to show all members of all objects in the system because the context of object is not yet specified.

In essentially every domain one has to make decisions and perform calculations. So another set of constructs that is hard to avoid is arithmetic, logcial and and comparison operators (together with types like number or Boolean), again independent of their concrete syntax (text, symbol or graphical). Again, luckily, most SMEs remember those from school or university. So this is also not a big challenge. The same is true for the notion of a conditional, such as `if ...then` or `switch{case, case, case}`.

The reason why these two lines are grey for SMEs and not totally black is because the complexity of the expressions built with these language concepts should (and usually can) be kept lower for SMEs. For example, for complicated decisions, we can support graphical decision tables of various forms which are much easier to grasp than nested `if` statements or the equivalent of `switch` statements.

Several of the DSLs I have built require an understanding of the notion of parametrization. For example, in function-like constructs, the values passed into the function are mapped (by position or by name) to formal arguments in the function signature that are then used in the body of the function. Most SMEs have no problems with this – again, school experience – but some do. In my experience this is the threshold where the need for education and training starts (beyond building the shared understanding about the core concepts of a domain). A related concept is the notion of instantiation, where, usually, the instance has independent values for its state and its evolution. This is not taught in school, and it's not taught outside of computer science at university, so training is needed. On the other hand, many DSLs can do without instantiation which is why this box is a lighter shade of gray.

We're getting to more advanced concepts that are increasingly harder to grasp for many SMEs, but they are also not necessary in all DSLs (though unavoidable in some domains). The notion of specialization or subtyping is key here. While everybody understand subtyping intuitively ("an eagle *isa* bird *isa* animal *isa* living thing *isa* object"), many SMEs struggle with the consequences. Especially the mental assembly of everything that's in a subtype by (mentally) going through all its supertypes is hard: "we're not seeing the big picture" is what I often hear. Practice helps, but so can tools that optionally show all the inherited members inline in the subtype's definition.

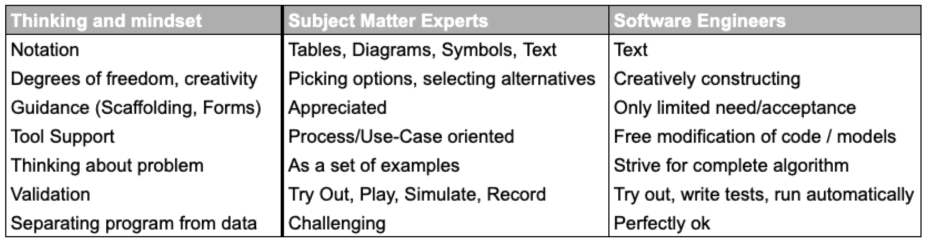
For complex subject matter – the tax stuff comes to mind – the models created with the DSLs become larger, and complexity often rises along with the size. Notions like delineating module boundaries, explicitly defined interfaces, reduction of unnecessary dependencies, and more generally, cohesion and coupling become an issue. Most SMEs struggle here. But on the other hand, 95% of the work of an SME can proceed without caring about these things, except during initial design or downstream review phases, where SD or guru involvement is often needed so sort these things out.

The final ingredient in region 4 is discovering and then defining new abstractions. This is usually not the strong suit of SMEs. Those that are good at it are usually the gurus we talked about before, or they have been assimilated wholly but the software development team. But luckily it is quite rare that SMEs are required to define new abstractions, because those that are relevant in the domain should be available first-class in the DSL – or retrofitted for the next version once the need becomes obvious with the help of SDs.

5.2 Different Emphasis

In my experience, (most) SMEs prioritize the features of languages and IDEs differently from (most) developers. In this section we'll look at some of the more prominenet differences, Fig. 4 summarizes them.

Notation Developers prefer textual notations, both for their conciseness, but also for reasons of homogeneity with regards to storing, editing, diffing and merging programs. SMEs, in contrast, tend to emphasize readability and fit of the notation with established representations in the domain (e.g., tables in the tax law documents) over these efficiency concerns. Therefore, if you can build



Thinking and mindset	Subject Matter Experts	Software Engineers
Notation	Tables, Diagrams, Symbols, Text	Text
Degrees of freedom, creativity	Picking options, selecting alternatives	Creatively constructing
Guidance (Scaffolding, Forms)	Appreciated	Only limited need/acceptance
Tool Support	Process/Use-Case oriented	Free modification of code / models
Thinking about problem	As a set of examples	Strive for complete algorithm
Validation	Try Out, Play, Simulate, Record	Try out, write tests, run automatically
Separating program from data	Challenging	Perfectly ok

Fig. 4. SMEs make different trade-offs than software engineers regarding the languages and tools they want to work with.

DSLs that are more diverse in notation – and not just colored text with curly braces and indentation – SME buy-in is usually easier to get.

Selecting vs. Creating Developers love the creative freedom of coming up with an algorithm and crafting their own suitable abstractions from small, flexible building blocks. SMEs – including because of their often limited abstraction skills – prefer picking from options and selecting alternatives. In my experience they accept if they have to read a bit more documentation that (hopefully!) explains what the different options or alternatives mean. So DSLs often rely on first-class concepts for the various needs of the domain, even if this requires the users to first understand what each of them means. The approach is usually also benefitial for domain-related semantic analysis (more first class content makes it easier to analyze programs) and it's easier to have a nice notation (because you can associate specific notations with these first-class concepts). In contrast, programming languages emphasize orthogonality and flexibility of their first-class concepts.

Guidance A related topic is guidance. Developers are happy with opening an empty editor and starting to write code. Code completion guides them a little bit. SMEs prefer more guidance, almost to the point where skeleton programs are pre-created after selection from a menu. DSLs that feel like a mix between a form-based application and program code seem to be particularly appreciated by many SMEs.

Tool Support Taking this further, SDs prefer a toolbox approach, where the tool provides 5 million different actions to do things. It's the developer's job to use each action at the right time, in the right way. SMEs are more use-case oriented. They want tool support for their typical process steps, and specific tool support for each. We have built DSLs that included wizard-like functionality in the IDE where using the wizard required more input gestures than just code-completion supported typing. Still the wizard was preferred by the SMEs.

Thinking about problems It's almost a defining feature of programmers that they think about a problem (and its solution) as a complete algorithm that can cover all possible execution paths. Sure, tests then validate specific scenarios, but developers *think* in algorithms. SMEs often think in terms of examples first, and sometimes exclusively. For example, it is easier for them to deal with a (hopefully complete) set of sequence diagrams rather than with a state machine. In terms of DSL design this means that more emphasis on case distinction where the various scenarios can be specified separately (even if this incurs a degree of code duplication) is often a good idea.

Validation Most developers are good at writing tests, writing them against APIs on a relatively small-size unit, and then running these tests automatically all the time. SMEs often think of validation more in terms of "playing with the system". They prefer "simulation GUIs" over writing repeatable tests as (a different kind of) program. So build those simulators first, and then allow the simulator to record "play sessions" and persist them as generated test cases for downstream reexecution.

Recipe vs. Execution A program is a recipe which, when combined with input data, behaves in a particular way. That behavior depends on the input data. So whenever SDs write code, they continuously imagine (and sometimes try our or trace with the debugger) how the program behaves for (all possible combinations of) input data. Many SMEs are not very good at doing this. One reason why Excel is so popular is because it does not make this distinction between program and data: the program always runs (or, alternatively, a spreadsheet never runs, it just "is"). So anything from the universe of live programming is helpful for DSLs.

Despite these differences, there are lots of commonalities as well. Both communities want good tools (read: IDE support), relevant analyses with understandable and precise error messages, refactorings and other ways to make non-local changes to (potentially large) programs, low turnaround time plus various ways of illustrating, tracing and debugging the execution of programs. However, while software engineers are often willing to compromise on these features if the expressivity of the language is convincing, SMEs usually will not.

6 Where and how can SMEs learn

So where and how can SMEs learn the skills from the SME column of Fig. 3?

In school and at university Ideally everybody should learn these basics in school and at university. While programming in the strict sense should be limited to computer science or software engineering curricula, this "SME'ing" should be mandatory for everybody, just like reading, writing or math. Of course such courses shouldn't just teach Java or Python. They should emphasize the specific skills of "thinking like a programmer" with a range of dedicated and diverse languages and tools.

Programming Basics Course A few years ago, based on the need to educate and trains some SMEs, I created a course called Programming Basics [6] that teaches these basics step by step. It starts with simple values as cells of spreadsheets and then covers expressions, testing, types, functions, structured values, collections, decisions and calculations as well as instantiation. The course uses different varieties and notations for many of these things to try and emphasize the conceptual aspects. It is built on MPS and KernelF, and allows extension and customization on language level towards particular DSLs. We are working on a way to get this into the browser for easier access.

Hedy Language Felienne Hermans has built Hedy [3], a gradual programming language. The goal is to teach "normal people" the basics of programming with a language that grows in capability step by step, with the need for each next capability motivated by user-understandable limitation in the previous step. Ultimately, the full stage of Hedy is Python. Hedy is free and works in the browser.

Computational Thinking In the 2000s, a community of software engineers came up with the term compuational thinking [2] as the "mental skills and practices for designing computations that get computers to do jobs for people, and explaining and interpreting the world as a complex of information processes." So the idea is kinda similar to what I am advocating, although the relationship to DSLs and subject matter is missing. Computational thinking has been critizied as being just another name for computer science; but my discussions in this paper should make clear that there's a big difference between computer science and that thing SMEs should do.

7 Wrap Up

It is almost not worth saying because it is obvious: almost all domains, disciplines, professions and sciences are becoming computational to an increasing degree. And market forces require companies – especially those in the traditional industrial countries – to become more efficient. I am confident that providing "CAD programs for knowledge workers", i.e., DSL, tools and automation, is an important building block for future success.

With the comparison of programming and "that thing SMEs should do" in this paper I hope to have made clear that everybody *doesn't* have to become a programmer. But: everybody has to be empowered to communicate the subject matter of their domain precisely to a computer (using DSLs or other suitable tools). And therefore, everybody has to learn to think like a programmer at least a little bit, enough to be able to understand and work with the things in the SME column of Fig. 3. And we software engineers have to adopt this subject-matter centric mindset and develop languages and tools that are built in line with the SME preferences in Fig. 4.

References

1. D. T. Administration. Challenges of the dutch tax and customs administration (video). `https://www.youtube.com/watch?v=_-XMjfz3RcU`, 2018.
2. P. J. Denning and M. Tedre. *Computational thinking*. MIT Press, 2019.

3. F. Hermans. Hedy, a gradual programming language. `https://hedy-beta.herokuapp.com/`, 2020.
4. itemis AG. The business dsl: Zurich insurance. `https://blogs.itemis.com/en/the-business-dsl-zurich-insurance`, 2019.
5. M. Voelter. Fusing modeling and programming into language-oriented programming. In *International Symposium on Leveraging Applications of Formal Methods*, pages 309–339. Springer, 2018.
6. M. Voelter. Programming basics: How to think like a programmer. `https://markusvoelter.github.io/ProgrammingBasics/`, 2018.
7. M. Voelter. Why dsls? a collection of anecdotes. `https://www.infoq.com/articles/why-dsl-collection-anecdotes`, 2020.
8. M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, 18(4):2507–2530, 2019.
9. M. Völter. The design, evolution, and use of kernelf. In *International Conference on Theory and Practice of Model Transformations*, pages 3–55. Springer, 2018.