Programming vs. That Thing Subject Matter Experts Do

Markus Voelter

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

Abstract. Allowing subject matter experts to directly contribute their domain knowledge and expertise to software through DSLs and automation is a promising way to increase overall software development efficiency and the quality of the product. However, there are doubts of whether this will force subject matter experts to become programmers. In this paper I answer this question with "no". But at the same time, subject matter experts have to learn how to communicate clearly and unambiguously to a computer, and this requires *some* aspects of what is traditionally called programming. The main part of this paper discusses what these aspects are and why learning these does not make people programmers.

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 engineers (SEs) when they implement it. 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 everyone involved. A better approach is to *empower* subject matter experts to capture, understand, and 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 executed either by interpretation or automatic 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.

2 Can SMEs use DSLs?

This paper is not about justifying the approach from a technical or economical perspective. I refer the reader to the Subject Matter First manifesto[1] Instead I want to focus on whether the SMEs are *able* to change from their typically imprecise, non-formal approach of specifying requirements using Word, Excel, User Stories or IBM Doors to this DSL-based approach.

Based on my experience over the years [10] my 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 DSLs have to become programmers? Do they have the *skills* to be programmers (hint: most do not), and do they *want* to become programmers (hint: most do not). But we still expect them to use "languages" and IDE-like tools. So:

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

I answer this question in Sec. 5. To set the stage we briefly discuss the domains in which the approach works (Sec. 3) and how languages and applications are typically architected in such scenarios (Sec. 4). We conclude the paper with a wrap up in Sec. 7.

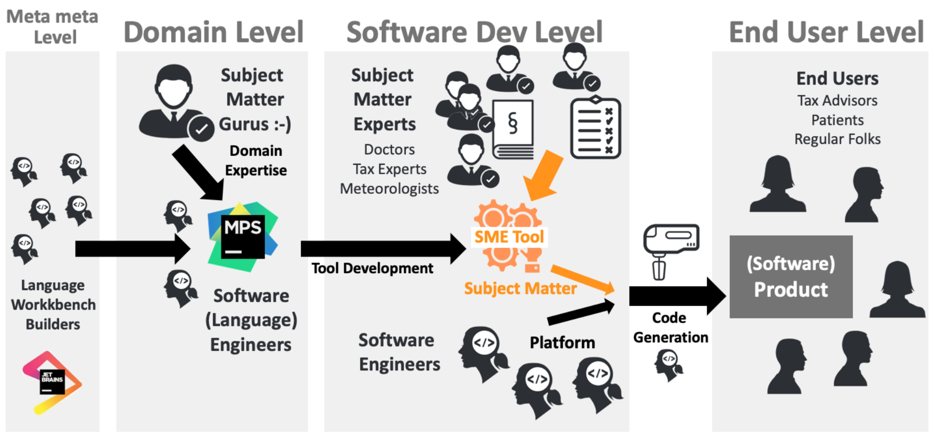


Fig. 1. The process from the SME's brain into software, based on tools and platforms developed by software engineers.

[1] `http://subjectmatterfirst.org`

3 (Where) Does this work?

Building the necessary language and IDE tooling and downstream automation requires investment, and this investment must pay off for the approach to make economical sense. Which is why this approach only works in domains that have the following characteristics. First, the subject matter has to have a minimum **size** and **complicatedness**. A consequence of this is often that there are people in the company who consider themselves **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 continue developing software in that domain for a reasonably long time. Finally, even though the subject matter as a whole r relevant for a long time, a degree of **evolution** or **variety** within the domain is usually needed for the approach to make sense. I 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 continuously evolving insurance products [4,7]. 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.

In healthcare, DSLs are used by medical doctors and other healthcare professionals to develop treatment and diagnostics algorithms that run as part of digital therapeutics apps [11]. My customer, Voluntis, 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.

In public adminstration, government agencies are certainly in it for the long run, while legislation for public benefits and tax calculation changes and evolves regularly. The agencies are full of experts who use DSLs [1] to disambiguate and formalize the law and its interpretations by courts. Similarly, the service providers who develop software for tax advisors have the same challenges and use DSLs as well [6,5].

In payroll, the 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 (and of course directly related). Service provides who develop payroll software therefore also employ whole departments full of experts and benefit from the use of DSLs [12].

For an overview over this approach and a couple of easily readable case studies, see my InfoQ article [10].

4 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 as the core of the system, usually

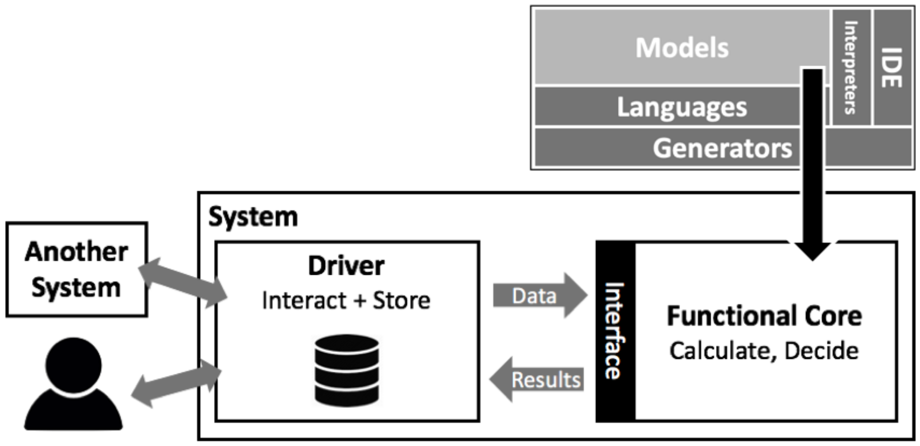


Fig. 2. Typical high-level systems architecture, where the models expressed with DSLs are transformed into code that forms the core of a larger application.

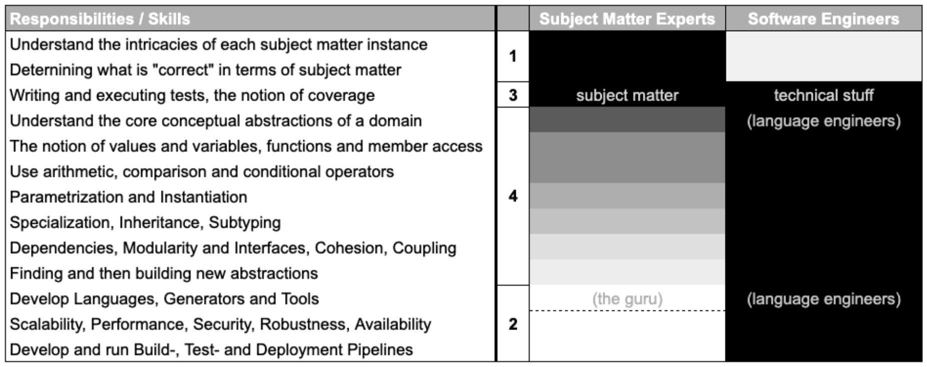
expressed with functional semantics, either via generation or via interpretation. That core implements a (manually defined) API that is used by a driver component to invoke the DSL-derived code. The driver interacts with users and other systems – potentially via additional architectural building blocks – and is often also responsible for managing and persisting state. Indeed, many of our DSLs are "funclarative" [8], where small, simple calculations are expressed with a functional-style language (so users do not have to care about effects at this level), and when things get more complicated, the DSL provides declarative first-class concepts to express those concisely without lots of low-level functional code.

In order to avoid reinventing the wheel with regards to the core functional expressions, the DSLs often embed (and then extend) a reusable language KernelF. I usually start by building a few of the domain-specific abstractions "around" KernelF. Then I iterate, building more abstractions, constraining away parts from KernelF that are not needed or replacing parts of KernelF with simpler abstractions. More details on KernelF and its use for DSLs can be found in [13].

If the DSL cannot be scoped to handling only the functional parts and thus has to manage state, I usually rely on variations of state machines. Generally it is a good idea to rely on established programming paradigms and DSL-ify them instead of trying to invent new fundamental paradigms.

5 Difference between programming and SME'ing

In this section we look at the work share between SMEs and SEs. We focus on what the SMEs do, because this paper is about the degree to which their activities resemble programming. We discuss the responsibilities of SEs mostly to contrast their work to that of the SMEs.



Responsibilities / Skills		Subject Matter Experts	Software Engineers
Understand the intricacies of each subject matter instance	1	■	□
Determining what is "correct" in terms of subject matter		■	□
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		▨	■
Use arithmetic, comparison and conditional operators		▨	■
Parametrization and Instantiation		▨	■
Specialization, Inheritance, Subtyping		▨	■
Dependencies, Modularity and Interfaces, Cohesion, Coupling		▨	■
Finding and then building new abstractions		▨	■
Develop Languages, Generators and Tools	2	(the guru)	(language engineers)
Scalability, Performance, Security, Robustness, Availability		□	■
Develop and run Build-, Test- and Deployment Pipelines		□	■

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".

5.1 Skills and Responsibilities

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

Region 1, SME only Region 1 is completely the responsibility of SMEs. They have to understand every particular example, case, situation, and exception of the subject matter they want the software system to handle. This is their natural responsibility, this is why they exist. SEs on the other hand 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, they write the tests. SMEs take full responsibility for what goes into test cases as well as for their completeness.

Region 2, SE only Let us move on to region 2, which is completely the responsibility of SEs. Setting up and operating automated CI pipelines that build and package the software and run tests is nothing the SME should be concerned with, except for being notified 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 this should be handled in platforms, frameworks and code generators – all fully the domain of SEs.

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 SEs. It might not be the domain of *all* of the software engineers, but maybe only of a certain specialization called language engineers who specialize in developing languages, IDEs, interpreters and generators. Also, a few of the most experienced subject matter experts – I 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.

Note that in order to be able to build the languages, the SEs have to understand the domain at the meta level. Together with the gurus, they have to understand how to describe the structures, rules, behaviors and other forms of knowledge in the domain. But they do not have to know all the instances which are subsequently expressed by the SMEs using the DSL. This distinction is crucial and is often perceived as a contradiction with the goal of the approach of separating the work of the SMEs and SEs.

Region 3, Testing Let us now look at region 3, one that apparently has full shared responsibility. However, in this case the illustration 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 the subject matter expressed with the DSL, whereas the SEs care about it *only* in the platform, frameworks, language implementation and generators. So they both have to understand testing, but there are no artifacts for which they have shared responsibility.

Region 4, shared skills This is the most interesting part: all the items in this region are native to programming and software engineering. So SEs care. But they are also relevant, to different degrees, to SMEs when they use DSLs. Let us 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 them. Maybe not every little detail (which is why the box is dark grey and not black) but sufficiently well to use the language. The SEs have to understand these abstractions as well, especially the language engieneers. Those who build the execution platform can usually 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 the domain for which the SMEs and the SEs co-create software is a major reason why DSLs are so useful here. The language definition, and its conceptual cousins, the core abstractions, are an unambiguous and clearly-scoped foundation for productive collaboration between the two communities. So when I write "xyz is the responsibility of SME/SE", it does not mean that there is not a *joint overall responsibility* of both communities together to deliver useful (in terms of subject matter) and robust (in terms of operational concerns) software.

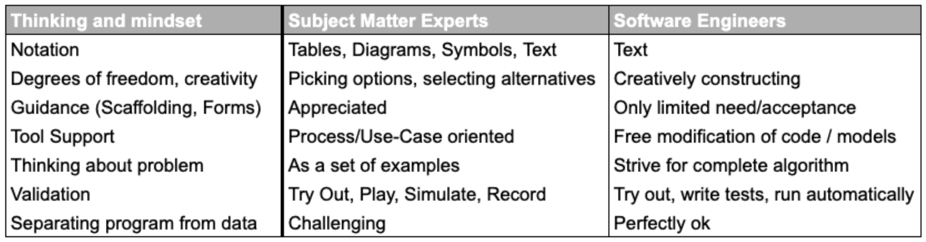
Back to region 4. No real-world subject-matter focused DSL I have ever seen can make do without understanding the notion of values, and some notion of functions (entities that produce new values from inputs). It does not matter whether we are talking specifically about (textual) functions, (Excel-style) decision tables or (graphical) dataflow diagrams. The good thing is that essentially everybody has come across these at school or at university, even though using functions to assemble larger functionality from pieces and the explicit use of types is new to many. In practice, teaching the use of functions, at least in limited complexity situations, is feasible.

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 ubiquitous. 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 results in less useful IDE support: with the latter syntax, one can easily scope the code completion menu to members of `object` because users write the object first. In contrast, with the former syntax, the code completion for the member has to show *all* members of *all* objects in the system because the context `object` is not yet specified.

In essentially every domain SMEs have to express decisions and calculations. So another set of constructs that is hard to avoid is arithmetic, logical and and comparison operators (together with types like `number`), as well as notion of a conditional, such as `if ...then` or `switch{case, case, case}` and the associated `Boolean` type – independent of their concrete syntax (text, symbolic or graphical). Once again, most SMEs have come across these operators at school or university, so using them is not a big challenge.

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 some form of `switch`-like statements.

Several of the DSLs I have built require an understanding of parametrization. For example, in function-like constructs, the values passed into the function are mapped (by position or by name) to parameters 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. Often parametrization 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 instantiation, where, usually, each instance has separate values for its state and can evolve independently. This is not taught in school, and it is 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.



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.

We are 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). The notion of specialization or subtyping is key here. While everybody understands 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 is in a subtype by (mentally) going through all its supertypes is hard: "we are 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 – tax calculation comes to mind – the models created with the DSLs become large, and complexity often rises along with size. Notions like delineating module boundaries, explicitly defined interfaces, reduction of unnecessary dependencies, and more generally, cohesion, coupling and reuse 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 big picture concerns, except during initial design or downstream review phases, where SE or guru involvement can help to sort things out. Considering it is only 5% of the total work, such involvement is usually feasible.

The final ingredient in region 4 is discovering and then defining new abstractions. This is often not the strong suit of SMEs. Those that are good at it are usually the gurus who help with language definition, or they have been assimilated wholly by 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.

5.2 Different Emphasis

In my experience, (most) SMEs prioritize the features of languages and IDEs different 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 code. 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 DSLs that are more diverse in notation – and not just colored text with curly braces and indentation – SME buy-in is usually easier to obtain.

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 – because of their often limited experience with building their own abstractions – prefer picking from options and selecting alternatives. In my experience SMEs usually accept that they have to read a bit more documentation that (hopefully!) explains what the different options or alternatives mean. Consequently DSLs often contain many 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 concepts makes it easier to analyze programs) and it is easier to have a nice notation (because you can associate specific notations with these first-class concepts). In contrast, programming languages emphasize orthogonality and composability of their (fewer) 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, SEs prefer a toolbox approach, where the tool offers lots and lots of actions and it is 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 workflows and process steps, and specific tool support for each. To give an extreme example: I 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 is almost a defining feature of SEs 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 that captures the superset of the sequence diagrams. In terms of DSL design this means that more emphasis on case distinction in which distinct scenarios are 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 for a relatively small-size unit, and then running these tests automatically continuously. 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 later automatic reexecution.

Recipe vs. Execution A program is a recipe which, when combined with input data, behaves in a particular way. The specific behavior depends on the input data. So whenever SEs write code, they continuously imagine (and sometimes try out 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 the program and its execution: 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 In my opinion, *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 should not 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 a group of SMEs, I created a course called Programming Basics [9] that teaches these concepts relevant to SMEs 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 concepts in order to try and emphasize the concepts. The course is built on the Jetbrains MPS language workbench[2] and KernelF, and allows extension and customization

[2] `http://jetbrains.com/mps`

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, when Hedy is fully developed, it is similar to 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 similar to what I am advocating, although the relationship to DSLs and subject matter is missing. Computational thinking has been critizised 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 so obvious: almost all domains, disciplines, professions and sciences are becoming increasingly computational. 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 economic success.

With the comparison of programming and "that thing SMEs should do" in this paper I hope to make clear that everybody *does not* 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.

Acknowledgements

Thanks to Yulia Komarov and Federico Tomassetti for providing feedback on an earlier version of this paper, as well as the anonymous reviewers of the ISOLA 2021 conference.

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. Y. K. Markus Voelter. Streamlining der Steuersoftware-Entwicklung bei DATEV mittels DomÃd'nenspezifischer Sprachen (slides). OOP Conference 2021, `http://voelter.de/data/presentations/oop2021-steuerDSLStreamlining.pdf`, 2021.
6. Y. K. Markus Voelter. Streamlining der Steuersoftware-Entwicklung bei DATEV mittels DomÃd'nenspezifischer Sprachen (Video). OOP Conference 2021, `https://youtu.be/q56wzLQkEho`, 2021.
7. N. Stotz and K. Birken. Migrating insurance calculation rule descriptions from word to mps. In A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, editors, *Domain-Specific Languages in Practice*, chapter 6, pages 165–194. Springer, 2021.
8. M. Voelter. Fusing modeling and programming into language-oriented programming. In *International Symposium on Leveraging Applications of Formal Methods*, pages 309–339. Springer, 2018.
9. M. Voelter. Programming basics: How to think like a programmer. `https://markusvoelter.github.io/ProgrammingBasics/`, 2018.
10. M. Voelter. Why dsls? a collection of anecdotes. `https://www.infoq.com/articles/why-dsl-collection-anecdotes`, 2020.
11. 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.
12. M. Voelter, S. Koscejev, M. Riedel, A. Deitsch, and a. Andreas Hinkel. A domain-specific language for payroll calculations: an experience report from datev. In A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, editors, *Domain-Specific Languages in Practice*, chapter 4, pages 93–130. Springer, 2021.
13. M. Völter. The design, evolution, and use of KernelF. In *International Conference on Theory and Practice of Model Transformations*, pages 3–55. Springer, 2018.