Reflections on the Lack of Adoption of Domain Specific Languages*

Federico Tomassetti[1] and Vadim Zaytsev[2,3]

[1] federico@tomassetti.me, https://tomassetti.me
Strumenta, Torino, Italy

[2] vadim@grammarware.net, http://grammarware.net
Raincode Labs, Brussels, Belgium
[3] Universiteit Twente, Enschede, The Netherlands

Abstract. Given all the different benefits that domain specific languages are reported to produce, why are they not a widely adopted practice? The essence of the question has roots in industrial experience of the two authors of this report, but it was put out as a discussion starter at OOPSLE 2020. During the discussion session itself, there were some possible reasons voiced and possibly related concerns expressed. In this report, we try to condense those in a short coherent text. Not claiming any generality and fully acknowledging the anecdotal nature of our evidence, we still think that such a conversation is useful to have within the SLE community.

1 Introduction

While this paper reflects the opinions of the authors, it has been strongly influenced by the helpful discussions at OOPSLE over the business and organisational problems causing domain specific languages (DSLs) not to be adopted. By the lack (or perhaps just dearth) of adoption here we mean that whenever software developers have a problem to solve, among the possible solutions to it, modelling the problem domain by means of creating a new domain-specific language or adopting an existing one, is rarely, if ever, the first option. Quite often it is being left out of this list altogether.

2 Advantages of DSLs

From research and practice, domain-specific languages are known to bring the following advantages:

- **domain-specific abstractions** [3, 7, 13, 17] to express commonly needed concepts even by non-developers, beyond what a library would allow [16];

* Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

– **domain-specific notations** [3, 7, 15, 17] that help expressing needed concepts in a readable and maintainable way, and the flexibility to change and adjust those notations as an inherent part of growing the language [13];
– **separation of concerns** stemming from the language focus: in a language that can only do one thing well, one is forced to write disjoint programs or models even for strongly coupled entities;
– **tool support** for type checking, editing [13], evolution [17] analysis, verification, optimisation, transformation [7], simulation and animation [18], derivation of dependent artefacts [14] (even though some sources explicitly complain that tool support for DSLs is noticeably worse than that for GPLs [3]);
– **conciseness** [3, 18] and **self-documentation** [2], to express the domain concepts in a concise manner that would make the resulting program or model intuitively understandable by domain experts;
– **productivity** [2, 3, 14] and **maintainability** [2] boosts, allowing domain experts to manipulate constructs expressed in a DSL, in a quick and natural way, drastically reducing maintenance costs;
– **reliability**, **testability**, **portability** and **safety** [2, 14] allegations based on making the language fit the chosen platform(s) and using its behavioural patterns in testing and verification to make sure it runs smoothly;
– **conservation and reuse of domain knowledge** [2, 13] without leaky abstractions [16] (during the OOPSLE discussion the main author of the latter cited paper even claimed that "understanding the domain is the most important side effect of building a DSL");
– **executability**, **liveness** and enabling debugging and experimentation with a system that models the domain in a highly interactive way [13];
– **involvement** and integration of domain experts beyond experienced programmers, into the development process [3, 13, 17], and their **collaboration** [17];
– the **lifespan** of a DSL is that of months or years [16] while general purpose languages live through decades notwithstanding their actual worth after such a deployment period, simply based on inertia of accumulated legacy.

Given the obvious enthusiasm about the potential of domain-specific languages within the (OOP)SLE community, it is discomforting to notice how low the adoption rate of DSLs appears to be, even in contexts which, according to the experience industries have with DSLs, would be a good fit for their adoption. In the remainder of the paper we try to summarise possible reasons without overselling DSLs as the silver bullet.

3 Adoption Problems of DSLs

During the discussion two main problems emerged, which seem to negatively affect DSL adoption:

1. Many software engineering professionals are unaware or oblivious of DSLs.

2. Many SE professionals who are aware of DSLs, perceive them as risky.

In the remainder of this section we discuss these specific evident symptoms of the problem. We will not be discussing the cause for these symptoms, nor trying to find other problems to make the list exhaustive.

3.1 Lack of Awareness for DSLs

As practitioners, we seem to agree that only a tiny minority of software development professionals are aware of DSLs. An anecdote we can report, is based on our experience working with a client on the development of a DSL. Before contacting us, an organisation had invested some effort internally to raise the level of abstraction in their software development process. They had the intuition it was possible to do so, in their context. However, the strategy to reach that goal, remained unclear until they were faced with the definition of the term "Domain Specific Language" and the implications around it. That term was encountered by them for the first time only after investing two years on this internal project.

While many practitioners have a total lack of awareness about DSLs, others have severe misconceptions about them. Often practitioners associate things unrelated to DSLs, to the misunderstood term. In other cases they are aware of only a certain kind of DSL, for example internal or embedded DSLs [10]. They assume those are the only possible type of DSLs and draw conclusions based on that. *"Look closely enough, and HTML and emojis become DSLs too"* [1].

It appears obvious that DSLs cannot be more widely adopted, until more software development professionals get a better understanding of them, or until the DSL community adopts a different terminology which is less alienating for developers.

3.2 Perceived DSL Risks

When there is a proper understanding of DSLs within an organisation, and they are aware of the benefits DSLs could bring to their specific situation, there is a set of perceived risks which could dissuade them from adopting DSLs.

In this section we are not making any statement about the validity of these risks, but only trying to identify risks as they appear to be perceived by potential adopters:

1. **Lack of competence.** Adopting DSLs is seen as risky because most companies simply do not have internal resources with the skills needed to design, implement, and maintain advanced DSL-based solutions. Such competence seems difficult to acquire on the market or to develop internally.
2. **Lack of established vendors.** It is perceived to be difficult to identify vendors providing language engineering services. In particular large organisations see the risk in the absence of large vendors capable of providing sufficient support for software language engineering projects. The majority

of providers of language engineering services are single freelancers or micro-consultancies. Raincode Labs is known to be the largest in the world with only around 50 compiler experts [9]. Working with vendors with single-digit number of employees poses a risk, because those vendors could not rapidly scale the support offered, and they do not offer sufficient guarantees for the long term.

3. **Lack of adoption.** It looks like a vicious circle, but DSLs are perceived as risky because they are not a mainstream technology. This poses a threat regarding future support of technologies related to DSLs. It also makes management more cautious in adopting a solution which is not common, as potential failures could be connected to the "unusual" choice of adopting DSLs. While "no one gets fired for buying IBM" [6], if a DSL-project proves unsuccessful, someone most probably will be. Mainstream general purpose languages typically enjoy a rich ecosystem with an arsenal of well-maintained tools and other forms of support, and even their discontinuation is well-announced and slow. Identifying and studying domain-specific ecosystems is a well-known challenge [11], but it is not uncommon for an ecosystem of a particular DSL to be confined to one company.
4. **Fear of lock-in.** An organisation could perceive the DSL route as risky, because of the lock-in on the developed languages, their supporting toolsets and/or the platform on which they rely. For example, a company which develops a DSL for the definition of digital therapeutics applications using JetBrains MPS [4] could later be locked-in in using the developed language(s), because of the complexity to switch to other technologies. Even if the company wanted to keep using the DSL, they may want to migrate to other platforms or language workbenches, but that could prove too difficult, in turn effectively locking them in the specific workbench and platform originally chosen (JetBrains MPS, in this example).
5. **Bad prior experience.** Organisations which are open to adopting non-mainstream solutions, may have adopted other technologies claiming some of the benefits claimed today by DSLs: 4GLs, RAD, etc. Because of this, they could attribute some of the already experienced drawbacks of those solutions to DSLs, perceiving DSLs as bringing the same risks as those previously tried solutions.

Most of these risks are far from being unfounded. It is indeed hard to become an accomplished expert in compiler services and software language engineering—even though only a very small team of such people is needed for any project or an organisation. It is indeed a big liability that DSLs can be abandoned simply because the one person making maintenance promises, decides to retract them and move on. It is indeed true that DSLs are not a universal solution to all the problems, and thus remain somewhat niche. We see the latter two as the most weakly supported by facts: while there is a lock-in chance, a well-designed DSL raises the level of abstraction enough to be separated from the technical implementation details, which means redeveloping an alternative toolset is just a matter of resources. (An example was given of TIALAA, short for "There Is A Life After

AppBuilder" [20,21], a project where even in the worst possible circumstances—for a badly designed 4GL which documentation was inaccessible—it was possible for a very small team to develop a full replacement compiler and debugger from scratch). The last item remains a topic of a larger discussion, which has no place here among DSL experts and fans, since all OOPSLE participants will obviously claim that DSLs are sufficiently different from solutions that preceded them.

4 The Term and Its Refinement

Since the term "Domain-Specific Languages" is problematic and overly general, there can be two paths to solve this problem: either refining it or abandoning it. Refinement will imply further classification around several dimensions, in order to clearly position each particular proposed solution within the overly large solution space, and to facilitate separate discussions around each of the dimensions, instead of wallowing in misconceptions. Examples of dimensions can be:

– **Internal/Embedded/External.** Internal DSLs are defined within the host language (e.g., Haskell's combinator libraries, jQuery and similar libraries for JavaScript, XML I/O in most languages). Embedded DSLs are defined naturally within an environment that was meant for significant extension (e.g., language workbenches, languages with powerful quoting like MetaOCaml, extensible compilers). External DSLs have foreign syntax and require specifically developed tooling (e.g., embedded SQL in COBOL and 4GLs, LINQ in C#, XPath and PCRE in all languages that support them). This particular definition of this dimension is heavily inspired by Renggli's work [10].
– **Horizontal/Vertical.** Horizontal DSLs, or languages specific to a horizontal domain, are those that are related to a specific type of activity that can be performed in any market sector (e.g., defining documents, modelling interactions, querying datasets). Vertical DSLs, sometimes also referred to as business DSLs, are languages specific to a certain market sector (e.g., insurance contracts, embedded systems, dance moves). This definitions was considered very early on, already by Kleppe [5].
– **Target Audience.** Some DSLs are intended to be powerful tools for programmers to do more with less code (if APL can be seen as a DSL for array processing, it will be this; any dialect of EBNF is also a perfect example: it allows a fully trained expert to fit a language definition on a page). Some other DSLs are specifically targeting domain experts without any background in computer science (many spreadsheets; languages made from established notations like the musical notation).
– **Ecosystem.** Anyone having trouble in Python, can post a question on the Data Science Stack Exchange website. Anyone debugging or optimising a SQL query, can post a question on the Database Administrators Stack Exchange website. Either of those get 25 questions a day, most of which are answered [12]. For some of the less known DSLs, the rate of questions is

lower, and they remain unanswered longer. Some do not even have any forum nor discussion board to go to. For the most obscure ones, even plain web search does not return any hits. A similar spectrum can be formulated for community size, tool support, compiler maintenance activities, etc—all the things that general purpose language users take for granted, are in high demand yet scarce availability for DSLs.

Examples of statements made aware of such dimensions:

– HTML is an *external* DSL, *targeting* developers, for a *horizontal* domain of hypertext markup. Building such a DSL and the corresponding tooling is an effort which requires significantly resources and it is intended to be fore-taken by the industry as a whole.
– Simple *internal* DSLs can be developed within a few hours, with the intent of making code more readable. They require minimal investment. They are *intended* mostly for developers, and in the majority of cases the management will not even be aware of their existence inside the organisation.
– *Vertical external* DSLs *directed* to non-developers, can be transformative for an organisation. They require the deployment of a significant effort and need the support of the whole organisation or a large unit, such as a department. They are typically multi-year projects with far reaching consequences in term of productivity. They require adequate planning as they need to be supported on the long term. This particular kind of DSLs are probably the least well-known by a broader audience. A relevant example of this particular kind of DSL is described by Völter et al [14].

If we consider these examples, we can see that they solve very different problems, they are intended to be used in different contexts, they require different resources to be implemented, they bring different risks with them, and they provide different benefits. In other words, it is hard to see them as an homogeneous category, from the point of view of the potential adopters. Some are expensive, some are not, some can be built in an afternoon, others take years, some need to be maintained for decades, others don't, for some it is easy to find competencies, for other is extremely difficult. As consequence hardly anything which can be stated for one sub-category of DSLs, can be extended to the whole category of DSLs. This could have contributed to the emerging of a very confused understanding of DSLs among many practitioners, and this could have in turn limited the diffusion as the idea, as it is blurred and confused.

In other words, the category "Domain-Specific Languages" seem too abstract for practitioners, while being perfectly useful for researchers. We could draw a similarity with other broad categories such as "means of transportation". While this term could be useful when discussing logistics or studying transportation, this would not be a term that many users would consider when looking for a solution to their problems, as it encompasses very different things such as bicycles, trains, tanks and aeroplanes, which are used in very different contexts to solve very different problems.

Hence, instead of refining it, we can try to completely abandon the term "DSL", and leave it behind. Alternatives to consider, are:

- **Fluent Interfaces** $\Longrightarrow$ Internal DSLs, APIs, library packages: they are characterised by requiring a very limited effort, introducing limited risks, and provide a limited value when compared to external DSLs, because of the lack of all advantages provided by tooling. While the developers of such DSLs themselves enjoy some comfortable IDEs, not all that confort and support is propagated well to the DSLs they create. Take `diagrams` [19] as an example: it is a popular Haskell package for creating vector graphics, and is being advertised as a DSL. Does it profit from the host language's strong type system? Absolutely! Does it have support for drag-and-drop functionality or for drawing tablet input? Absolutely not!
- **Technical Languages** $\Longrightarrow$ External DSLs for developers, often horizontal. These DSLs can have a very significant impact on the industry and typically their creation and development is fore-taken by several large partners. Examples of Technical Languages are SQL, HTML, and CSS.
- **Knowledge Systems** $\Longrightarrow$ External executable DSLs for non-developers, often vertical. This type of DSLs are typically developed by a single organisation or a consortium of organisations operating in the same vertical space. They require significant investments, and given they are intended for a specific vertical space, only a reduced number of users can benefit from them. Therefore the investment requested to each adopting organisation is significant. A key factor of this kind of DSLs is the creation of customised tooling: besides editors, interpreters, simulators, diagram generators, and other components are frequently developed and integrated in a comprehensive solution. This is the kind of DSLs least widely known: unlike other kinds, such DSLs are not typically shared outside the organisation which financed them. An example of such kind of DSL is the DSL created by the Dutch Tax and Customs Administration for the definition of tax calculations [8].

In Table 1 we present the relation between the DSL dimensions we discussed in section 4 and the DSL types we introduced here.

As an example of how DSL types can also be useful, consider the discussion on how risks affect the adoption of a DSL. Different types of DSLs that we have identified, are impacted very differently by the risks perceived:

- In the case of **Fluent Interfaces** the risks involved are limited, as the investments necessary, and the benefits provided.
- In the case of **Technical Languages**, these languages are typically tackled as large, industry-wide efforts. While the resources necessary to tackle such projects are very large, the effort is typically spread across multiple actors. The actors typically involved in these initiatives have the resources needed to counteract the risks involved.
- In the case of **Knowledge Systems**, the risks faced are more significant because the level of investment needed is high, and the number of adopters to

Internal (I)/ External (E)	**Horizontal (H)/ Vertical (V)**	**Developers (D)/ Anyone (A)**	**DSL Type**
I	H	D	Fluent Interface
I	H	A	*Fluent Interface*
I	V	D	Fluent Interface
I	V	A	*Fluent Interface*
E	H	D	Technical Language
E	H	A	*Knowledge System*
E	V	D	*Technical Language*
E	V	A	Knowledge System

Table 1. Relation between DSL dimensions and DSL Types (less common forms in italics)

share such risks is low. In addition to this, there is a problem specific to this kind of DSLs, and this the lack of examples which are publicly accessible. We believe that the community should concentrate its effort in mitigating the risks for this type of DSLs. More specifically:

- **Lack of competence** could be reduced by an effort in education. Under the umbrella of SLEBoK [22] a list of university courses teaching DSLs has been created. This could help fostering collaborations among teachers and possibly facilitate the creation of more courses on DSLs;
- **Lack of established vendors** is a real problem, to which there is not an immediate answer. Larger vendors would hopefully emerge as the adoption of DSLs is increased;
- **Lack of adoption**, is unfortunately a self-fulfilling prophecy;
- **Fear of lock-in** is a risk which can be reduced by working on interoperability and the adoption of common standards, or at least the definition of tools to translate among the most common formats;
- **Bad prior experience** could be counteracted by proper communication of the differences between DSLs and other similar technologies. Efforts to clarify how DSLs are presented are necessary, and in this paper we present some ideas to move in that direction.

5 Conclusion

Speaking pessimistically, we can say that as a community of language designers, we are failing to communicate all the possibilities and benefits that DSLs and the discipline of their engineering, can offer to practitioners. DSLs remain niche solutions to problems experienced by people who are both already aware of them and unafraid to venture towards them. In particular, *meta*programmers seem to be a good target audience for selling DSLs to.

In this paper, we tried to summarise the essence of the discussion that was taking place during OOPSLE 2020, the workshop on Open and Original Problems in Software Language Engineering. To substantiate some of the statements and to facilitate consumption of this text by external readers, we added some technical content, especially relating to the advantages of DSLs, that we collected in section 2 from available literature; the two adoption problems of DSLs, including a list of perceived risks of using them, in section 3; possible ways of improving the terminology by either refining or replacing the very term "DSLs", in section 4. Our hope is that claims made here, can later be validated experimentally, and proper corrections could be put in place by the community to improve the level of adoption of language engineering techniques in the broader field of software engineering.

Acknowledgement

We would like to thank all the participants of the discussion which was held as part of OOPSLE 2020. In particular, active participation in the discussion was noticed and appreciated from: Jurriaan Hage, Markus Völter, Ralf Lämmel, Friedrich Steimann, and Mathieu Acher. We are also grateful to the organisers and other attendees of the OOPSLE workshop.

References

1. Blasband, D.: The DSL Misunderstanding. https://www.linkedin.com/pulse/dsl-misunderstanding-darius-blasband/ (Jul 2017)
2. van Deursen, A., Klint, P., Visser, J.: Domain-specific Languages: An Annotated Bibliography. SIGPLAN Notices **35**(6), 26–36 (Jun 2000). https://doi.org/10.1145/352029.352035
3. Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J.P.: DSLs: The Good, the Bad, and the Ugly. In: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications. p. 791–794. OOPSLA Companion '08, ACM (2008). https://doi.org/10.1145/1449814.1449863
4. JetBrains: MPS: Meta Programming System. https://www.jetbrains.com/mps (2009)
5. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional (2008)
6. Lynch, P., Rothchild, J.: Beating the Street. Simon & Schuster (1994)
7. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys **37**(4), 316–344 (2005). https://doi.org/10.1145/1118890.1118892
8. MPS: Client: Dutch Tax and Customs Administration (DTCA). Tech. rep., JetBrains (2014), https://resources.jetbrains.com/storage/products/mps/docs/MPS_DTO_Case_Study.pdf
9. Raincode Labs. https://www.raincodelabs.com
10. Renggli, L.: Dynamic Language Embedding With Homogeneous Tool Support. Ph.D. thesis, Universität Bern (2010)
11. Serebrenik, A., Mens, T.: Challenges in software ecosystems research. In: Proceedings of the 2015 European Conference on Software Architecture Workshops. pp. 1–6 (2015). https://doi.org/10.1145/2797433.2797475

12. StackExchange: All Sites. https://stackexchange.com/sites?view=list (2020)
13. Völter, M.: Fusing Modeling and Programming into Language-Oriented Programming — Our Experiences with MPS. In: Margaria, T., Steffen, B. (eds.) Proceedings of the Eighth International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Part I. Lecture Notes in Computer Science, vol. 11244, pp. 309–339. Springer (2018). https://doi.org/10.1007/978-3-030-03418-4_19
14. Völter, M., Kolb, B., Birken, K., Tomassetti, F., Alff, P., Wiart, L., Wortmann, A., Nordmann, A.: Using Language Workbenches and Domain-Specific Languages for Safety-Critical Software Development. Software and Systems Modeling **18**(4), 2507–2530 (2019). https://doi.org/10.1007/s10270-018-0679-0
15. Völter, M., Visser, E.: Product Line Engineering Using Domain-Specific Languages. In: Proceedings of the 15th International Software Product Line Conference. pp. 70–79. IEEE (2011). https://doi.org/10.1109/SPLC.2011.25
16. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
17. Wegeler, T., Gutzeit, F., Destailleur, A., Dock, B.: Evaluating the Benefits of Using Domain-Specific Modeling Languages: An Experience Report. In: Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling (DSM). p. 7–12. ACM (2013). https://doi.org/10.1145/2541928.2541930
18. Wile, D.S.: Supporting the DSL Spectrum. Journal of Computing and Information Technology **9**(4), 263–287 (2001). https://doi.org/10.2498/cit.2001.04.01
19. Yorgey, B.: diagrams. https://archives.haskell.org/projects.haskell.org/diagrams/ (2015)
20. Zaytsev, V.: Parser Generation by Example for Legacy Pattern Languages. In: Flatt, M., Erdweg, S. (eds.) Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE). pp. 212–218. ACM (2017). https://doi.org/10.1145/3136040.3136058
21. Zaytsev, V.: An Industrial Case Study in Compiler Testing. In: Pearce, D.J., Mayerhofer, T., Steimann, F. (eds.) Proceedings of the 11th International Conference on Software Language Engineering (SLE). pp. 97–102. ACM (2018). https://doi.org/10.1145/3276604.3276619
22. Zaytsev, V. (Ed.): Software Language Engineering Body of Knowledge. http://slebok.github.io (2009–2020)