# The DSGA Model of DSL Design: Domain, Schema, Grammar, Actions

Vadim Zaytsev
Raincode, Belgium
vadim@grammarware.net

## Abstract

In the proposed talk the following two open problems of software language engineering are discussed: (1) the process of designing a domain-specific software language and (2) the method of teaching domain-specific language design.

The first problem is well-known and simultaneously well-hidden: each individual member of the SLE/DSL community has their own method of designing software languages and especially of implementing them, and as time goes by, gets used and attached to this method, and quite understandably so. There are very few attempts to compare the methods, and even they usually stop at feature modelling [3]. Meanwhile, all such methods receive harsh critique from the industry for being limiting upfront, because in practitioners' eyes domain specificity starts with embracing the domain, eliciting domain knowledge and communicating with domain experts, where demands like "we will do it with Haskell" or "it will work within our Smalltalk environment" on the requirements engineering stage are premature and unrealistic, since real solutions should integrate into clients' infrastructure, not the implementers' [1].

The second problem connects tightly into the first one, but adds another layer of complexity. Where do we begin to teach DSL design? What is the right order of topics? How can we do it in a technology independent way? More importantly, how do we teach non-computer scientists to design languages? (Because with all due respect, there are many more non-programmers and non-engineers in the world than otherwise). Should we put semantics first, as suggested by Erwig and Walkingshaw [4], or explain that semantics is irrelevant to the language structure, per Chomsky [2]? Should we attribute the abundance of general purpose languages to their domain specificity (and therefore cluster them accordingly), following Völter et al. [13], or focus on the fact that DSLs tend to evolve with features from general purpose languages, following Tratt [12], or combine both views? Is it even important? Is Wirth right in saying that language simplicity comes predominantly with modularity [14], or is Hoare right in saying that language simplicity and modularity are different and conflicting, with simplicity being the winner in importance [7]? There are numerous claims and assumptions like these, including many that are "common knowledge" but cannot be traced to publications and experiments, and there is no studybook yet.

Last year I was invited to give a workshop on language design at HDSA'16, a summer academy for designers (`http://summer.hackersanddesigners.nl` [5], "Bugs, Bots and Bytes"). The workshop lasted one day. It included introductory lectures on the history of software languages from Ada Lovelace' notes [10] to von Neumann and the Goldstines' flow diagrams [6] to Grace Murray Hopper's first automated translators [8] to milestone languages from the design perspective like APL, LOGO, ALGOL-68, UML and examples of DSLs for sheet music, electronic circuitry, pianola plays, etc. That part was entertaining but not innovative to be discussed in detail. The concluding part was hands-on designing, which was also straightforward to execute, assuming the part in between did a good job at explaining what constitutes language design in a broad sense. The target audience mostly consisted of product designers, graphic designers, web designers and very few people with computer science background, 22 in total.

The following model was developed to explain DSL design to non-DSL designers. There are these four components of language design, that must be thought of and about, and one chooses the component to start with depending on the particular circumstances:

- **Domain**: what will the language be used for? Algorithms ("programming language")? Markup? Data? Constraints? Music? Dance? Finance? Food? What problems will it help solving? What are the fundamental concepts in this domain? What are their properties and interrelations?

*DSLDI'16* 31 Oct 2016, Amsterdam, The Netherlands
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

- **Schema**: what are sentences of the language, conceptually? Lists? Sets? Trees? Graphs? Tables? Looking inside a sentence, what is there? Are there different kinds of sentences?
- **Grammar**: how do we write sentences down? What alphabet is used? How symbols are constructed in it? Text? Table? Diagrams? Unicode? Colours? Sounds? Gestures?
- **Actions**: what kind of actions do we want the machine perform while executing a program in our language? Is it even intended for automated processing? How do words and sentences correspond to actions?

In classic SLE terms, the **Domain** part is the closest to a domain model or a domain-specific ontology; the **Schema** corresponds to a database schema, a metamodel, abstract syntax, an algebraic data type; the **Grammar** is indeed a grammar in the narrow sense of concrete syntax definition, and **Actions** are about the semantics, virtual machine and similar underlying implementation details and anything else related to runtime and execution.

Several typical ontology visualisation techniques were shown, taken from the survey by Katifori et al. [9], but most participants recognised the familiar mindmap view and settled for it. Grammars were exemplified by BNF-like textual notation, syntax diagrams and the gesture vocabulary from *Make It So* [11].

Each participant pitched their idea for a language, and they have all divided themselves in groups around the ideas that seemed most promising. Most started with either a **Domain** or a **Grammar**. The examples of the domain-driven design process were groups that designed languages for communication between designers and coders (in the context of a web development company) or for communicating with legacy computers (in the context of a museum trying to preserve cultural heritage artefacts requiring obsolete hardware). The examples of the grammar-driven design process included languages written exclusively with emojis or made of movements of riding a bicycle. An example of a schema-driven process was a language which program was supposed to be painting, and an example of an action-driven process was a language to order pizza.

At DSLDI I would like to have a discussion around the Domain-Schema-Grammar-Actions model of DSL design, touch on related topics such as software language design and the approaches to teaching it, and consider the model's viability as well as its alternatives.

## References

[1] D. Blasband. *Rise and Fall of Software Recipes*. Reality Bites Publishing, 2016.

[2] N. Chomsky. *Syntactic Structures*. Mouton, 1957.

[3] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In M. Erwig, R. F. Paige, and E. Van Wyk, editors, *Proceedings of the Sixth International Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[4] M. Erwig and E. Walkingshaw. Semantics First! — Rethinking the Language Design Process. In A. M. Sloane and U. Aßmann, editors, *Revised Selected Papers of the Fourth International Conference on Software Language Engineering (SLE)*, volume 6940 of *LNCS*, pages 243–262. Springer, 2011.

[5] S. Gildemacher, J. B. Graves, and A. Groten, editors. *Post-proceedings of HDSA 2015: About Bugs, Bots & Bytes*. De Punt, 2015.

[6] H. H. Goldstine and J. von Neumann. *Planning and Coding of Problems for an Electronic Computing Instrument, Part II, Volume 1*. Institute for Advanced Study, Princeton, NJ, 1947. https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf.

[7] C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[8] G. M. Hopper. Automatic Programming: Present Status and Future Trends. In *The 10th National Physical Laboratory Symposium on Mechanisation of Thought Processes*, pages 155–200, 1959.

[9] A. Katifori, C. Halatsis, G. Lepouras, C. Vassilakis, and E. G. Giannopoulou. Ontology Visualization Methods — A Survey. *ACM Computing Surveys*, 39(4), 2007.

[10] A. Lovelace. Notes by the Translator. In *Sketch of the Analytical Engine Invented by Charles Babbage, Esq.*, volume 3 of *Scientific Memoirs*, 1843.

[11] N. Shedroff and C. Noessel. *Make It So*. Rosenveld, 2012.

[12] L. Tratt. Evolving a DSL Implementation. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *LNCS*, pages 425–441. Springer, 2007.

[13] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[14] N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.