International Journal of Soft Computing 10 (5): 340-343, 2015
ISSN: 1816-9503


# Design of the Modern Domain Specific Programming Languages

R.A. Valiyev, L.A. Galiullin and A.N. Iliukhin
Branch of Kazan Federal University in Naberezhnye Chelny, 10A Sujumbike Avenue,
The City of Naberezhnye Chelny, The Republic of Tatarstan, Russian Federation

**Abstract:** In the modern industry of the software design the new paradigm, the new approach Language-Oriented Programming (LOP) becomes more and more popular. LOP is such approach to programming that is based on creation of the Domain-Specific Language; DSL) for solution of tasks within a specific subject area. In the LOP, a programmer firstly creates a single or a few DSL for solution of a particular set of tasks and then uses the created DSL by the software system design. They distinguish the two main kinds of the domain-specific languages: external (external DSL) and internal (internal DSL or embedded DSL) ones. The external DSL feature their own syntax separated from the core language of the application. The internal DSL use as the basis a general purpose programming language but differ through using a specific subset of capabilities of this language within a particular style. The study provides the review of the modern means (languages, platforms, development environments) allowing creating both external and internal DSL. It shall be noted that one of the most important issues by creation and further use of DSL is availability of a language workbench. The language workbench represents specialized integrated development environment, IDE for identification and creation of DSL. It is the complexity of creation of the language infrastructure required for implementation of DSL of different kind and operational comfort being one of the reasons of the small-scale use during commercial development of the software where the high developers' performance is crucial. In this review, special emphasis is laid on technologies allowing ensuring support of development in the LOP-style by means of the language workbench.

**Key words:** Programming languages, software, design, development, information architecture, information systems, programming techniques (methods), domain-specific programing

## INTRODUCTION

The process of creating a language with the own syntax, i.e., an external DSL may be represented as a sequence consisting of three steps:

- Determining the semantic model
- Determining the syntactic model (abstract and concrete syntax)
- Specifying the transformation rules (the rules according to which an abstract representation is transformed into executable one)

If for determination of the concrete language syntax and assignment of the transformation rules there are the ready-to-use tools of different kind starting from the bundle of programs lex+yacc integrated in the POSIX standard for generation of the lexical and stylistic analyzer, respectively and ending with the modern tools for automation of the metacompiler design for example ANTLR (Porkolab *et al*., 2015).

For determination of the Semantic Model of the language (Halupka, 2015) (the part of the language describing semantics of the domain area or a particular aspect of the system for configuring of which of the external DSL is designed there is no special software available; each team of developers solves the issue of representation of the DSL semantics independently, usually by describing the meta model of the language in one of the programming languages (as a rule, the general purpose programming languages). Besides, there are no means of the syntactic representation of the DSL semantics. The problem of such representation is also solved by a particular team of developers independently.

Because of the absence of the language workbench for support of the semantic language model and support of the semantic representation by means of the Syntactic Model the expenditures on design of the external DSL are increased. The external DSL themselves become self-contained languages for solution of narrow tasks and in practice it is almost impossible and unreasonable to use them repeatedly for solution of tasks in other domain areas (related, similar in the nature or content).

**Corresponding Author:** R.A. Valiyev, Branch of Kazan Federal University in Naberezhnye Chelny, 10A Sujumbike Avenue, The City of Naberezhnye Chelny, The Republic of Tatarstan, Russian Federation

The development environment that would support and facilitate scripting in the external DSL is usually designed either from scratch or as a plug-in to the already existing modern IDE.

Almost all modern development environments (for example, Eclipse IDE, Microsoft Visual Studio, etc.) feature a flexible plug-in architecture and allow adding support of the new programming languages.

## SUPPORT TOOLS FOR DESIGN OF THE INTERNAL DSL BASED ON THE GRAMMATICAL SYSTEMS OF THE GENERAL PURPOSE PROGRAMMING LANGUAGES

In the simplest case by design of an internal DSL one of the general purpose programming languages is chosen as a core language on the basis of which the library being kind of add-in to the language (Matsui and Aida, 2015) is developed that is further used in a specific style as a rule for managing particular aspects of the software system being designed.

It shall be understood that as opposed to the external DSL by design of the internal DSL the grammar of the core language impose restrictions on the expressive means of the language. The less flexible the grammar of the core language is the less convenient and efficient the internal DSL will be. Thus, the expressive power of the core language shall correspond to the area and method of application of the internal DSL designed on its basis.

By design of the internal DSL they most frequently rely on the grammar of the modern general purpose programming languages providing flexible options that allow creating convenient DSL. For example, these are such languages as Ruby, Python, Scala, C#, F#, Haskell. One may notice that in the list of such languages the multi-paradigm languages prevail that usually inherit the expressive power from a few, as a rule, unrelated languages. Due to such combination of various capabilities of the language grammar we obtain an efficient flexible tool for design of the internal DSL.

One should also emphasize the efficiency of use of the expressive means of the functional programming paradigm; however, it is possible to create the convening internal DSL even in non-functional languages with static type checking for example in C++ with the use of the templates mechanism.

By choosing a modern general purpose programming language as the basis for creation of an internal DSL, we immediately get a ready set of tools for the development support modern IDE supporting the core language.

Thus, creating the internal DSL, we sacrifice the complete freedom of the grammar assignment staying within the frameworks of the core language grammar, however, at the same time we are able to use the modern integrated development environments.

## DSL DEVELOPMENT SUPPORT TOOLS BASED ON THE PROGRAMMING LANGUAGES AND METHODS WITH CONFIGURABLE GRAMMAR

Another approach to DSL design (in point of fact, internal DSL) is the use of the programming languages with configurable syntax, i.e., oriented towards the meta programming techniques. Such approach is called the 'extensible programming'; it had been actively developing in 1960, then its development was suspended and the interest in this approach aroused again in the 21th century (Gibbs *et al*., 2015).

'Extensible programming' is a programming style oriented towards the use of the mechanisms of extension of the programing languages, translators and runtime environments (Matsui and Aida, 2015).

## LANGUAGES FOR THE DSL DEVELOPMENT SUPPORT WITHOUT TEXT GRAMMARS

The syntax of all modern general purpose programming languages (including those provided as examples above) is based on the text grammars. These grammars have one significant defect: at the attempt of the grammar extension it may become ambiguous, i.e, a few interpretations of the same string of the source code in such extended language may exist (Lachgar and Abdali, 2015). This issue is thrown into especially sharp relief at the attempt of combining a few extensions of grammar of the same language that are separately being unambiguous but by combining, thereof the resulting grammar may lose its unambiguousness and make the further use thereof impossible.

The issue of the grammar ambiguity may be solved by means of refusing the use of the text grammar as such in this case the program will be set as a sample of a particular syntactic metamodel. A software metamodel is usually represented as an abstract syntactic tree (Buzzi *et al*., 2014). By using such approach the design of a new DSL is reduced to assigning the DSL metamodel by means of the core language. The vivid example of the use of such approach to the DSL development is the Lisp language family: Common Lisp (Ali *et al*., 2014), Scheme (Kalnins *et al*., 2014), clojure (Common Lisp was already mentioned above as it may be used within the frameworks of the 'extensible programming' method as well).

Lisp (from Eng. LISt Processing language) is the family of the programming languages the programs and data in which are represented by the systems of the linear lists of symbols. Lisp is the second high-level programming language in the history (after FORTRAN) used up to now. In the beginning, Lisp was created as the tool for modeling various aspects of artificial intelligence, however, later on the area of the language application was extended.

Due to the minimalistic own syntax of the language, its dynamic typing, developed system of the compiling macros, the culture of incremental development and other features of the languages of the Lisp family, they have no superior in the speed and convenience of design of the internal (integrated) DSL. However by design of a DSL in Lisp the developers face another problem the complexity of creation of the language infrastructure required for implementation of the new DSL and operational comfort.

## MODERN LANGUAGE WORKBENCH FOR SUPPORT OF THE DSL DEVELOPMENT WITHOUT TEXT GRAMMARS

In point of fact, the language workbench represents tools that not only facilitate creating the own DSL but ensure the support thereof in the style of the modern intelligent development environments as well offering opportunities for construction of the modern IDE for the languages being designed. As the result, the programmers that will deal with the DSL scripting will be provided the same tool support as the programmers coding in the general purpose programming languages (C/C++, Java, C#, etc.). The development environments for DSL will provide such options without which the modern commercial software design is impossible, for example:

- Code completion, code generation, refactoring tools
- Tools for convenient debugging of the DSL-scripts
- Version control tools
- Unit and integration testing tools
- Reverse engineering tools

## SUMMARY

The methods described in this study have been used by development of the application for an automated information system for diesel engine testing (Zubkov and Galiullin, 2011; Biktimirov *et al*., 2014).

## CONCLUSION

In conclusion of the review, it may be noted that the number of languages and tools allowing not only to create DSL but providing efficient methods of updating the DSL as such and supporting development in the languages designed is rather small. After all, only JetBrains MPS and partially Helvetia (but as opposed to JetBrains MPS this system is also not developed) represent the modern language workbench covering not only the design stages but the stages of the DSL operation and updating as well without which the efficient use thereof in the software design industry can not be imagined.

The use of the language workbench requires the detailed study and gain in experience in application of such tool by the DSL design. The JetBrains MPS product features a relatively high barrier to entry which hinders the wide spread and application thereof. Besides, another weakness of the modern language workbench is the absence of standards and possibility of transfer of the DSL designed between different environments. Thus, having started to design DSL in JetBrains MPS, a language developer becomes a hostage of this workbench, since it does not allow exporting the DSL being designed. This is determined by the absence of any commonly accepted formats for DSL presentation and the significant difference in approaches to the DSL design.

The emphasis shall be laid on the fact that none of the tools for the DSL design does not allow to present the semantic model of the language to the full extent and adjust the representation of the language semantics based on the syntax All tools are focused mainly on presentation of the DSL text grammar or on presentation of the meta model under which an abstract syntactic tree is usually meant. However, M. Fauler reasonably notes that the DSL semantic models, as a rule, differ from an abstract syntactic tree. If a syntactic tree corresponds to the DSL script structure to the form (for example, in JetBrains MPS) then the semantic model of the language in its turn is based on how the script information will be processes this is the meaning, content. A Semantic Model shall reflect the essence, specifics of the domain area. And it is presence of a Semantic Model that constitutes one of the essential differences of working with DSL from working with the general purpose programming languages. It may be assumed that lack of efficient tools of the DSL semantics presentation on the syntax basis hinders the wide use of DSL in the software design.

## REFERENCES

Ali, K.M., G.G. Messier and S.W. Lai, 2014. DSL and PLC co-existence: An interference cancellation approach. IEEE Transactions on Communications, Art. No. 6877618, 62 (9): 3336-3350.

Biktimirov, R.L., R.A. Valiev, L.A. Galiullin, E.V. Zubkov and A.N. Iljuhin, 2014. Automated test system of diesel engines based on fuzzy neural network. Res. J. Applied Sci., 9 (12): 1059-1063.

Buzzi, S., C. Risi and G. Colavolpe, 2014. Green and fast DSL via joint processing of multiple lines and time-frequency packed modulation. Phys. Commun., 13 (PC): 99-108.

Gibbs, I., S. Dascalu, F.C. Harris, Jr., 2015. A separation-based UI architecture with a DSL for role specialization. J. Syst. Software, 101: 69-85.

Halupka, I., 2015. DSL for grammar refactoring patterns. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8606: 446-458.

Kalnins, A., L. Lace, E. Kalnina and A. Sostaks, 2014. DSL based platform for business process management. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8327 (LNCS), pp: 351-362.

Lachgar, M. and A. Abdali, 2015. Modeling and generating the user interface of mobile devices and web development with DSL. J. Theoretical and Applied Information Technol., 72 (1): 124-132.

Matsui, A.A.M. and H. Aida, 2015. A DSL for contract-centric compatibility assessment in distributed services. J. Infor. Processing, 23 (1): 41-57.

Porkolab, Z., A. Sinkovics and I. Siroki, 2015. DSL in C++ template metaprogram. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8606: 76-114.

Zubkov, E.V. and L.A. Galiullin, 2011. Hybrid neural network for the adjustment of fuzzy systems when simulating tests of internal combustion engines. Russian Eng. Res., 31 (5): 439-443.