Domain Specific Languages in Practice

Ivo Damyanov
South-West University
Blagoevgrad
Bulgaria

Mila Sukalinska
South-West University
Blagoevgrad
Bulgaria

ABSTRACT
Over the last three decades, an increasing number of languages used for designing and developing software have been created. Software developers gain the benefits of combining multiple programming languages and paradigms in application development, as a result the so-called language engineering approach can be outlined. It involves Domain-Specific Languages (DSLs) and automatic code generation.

This paper offers a brief review of the use of DSL as a modeling and programming language and it tight connection with automatic code generation. The evolution of the developed software product requires evolution of the domain-specific language as well. Some of the risks of abandoning of DSLs during development are discussed.

Keywords
Metaprogramming, DSL, code generation, language engineering, modeling.

1. INTRODUCTION
The rapid improvements in software development tools over the last decade allows software developers to increasingly use domain-specific languages and automatic code generation. These two contrivances are examples of the so-called metaprogramming. Metaprogramming itself is a paradigm in which programs are designed to read, generate, analyze and transform other programs or modify themselves while running.

Domain-specific languages are adjusted to a particular domain and provide notations close to it [1]. Based on the features of the problem domain they improve the communication between developers and domain experts.

One of the first detailed publications on domain-specific languages was published by Jon Bentley in 1986. He referred to them as little languages [2]. DSLs as little languages are tightly bound to a specific domain and their expressive power significantly differs from that of General Purpose Languages (GPLs). However, DSLs can improve development time and program correctness.

In 1994 Martin Ward [3] describes the concept of problems solving with the implementation of domain-specific languages and called this paradigm language-oriented programming.

There are various techniques introduced to manage the complexity of the application development process. One of these is the so-called Domain-Driven Design (DDD) – introduced by Eric Evans [4]. He points out the fact that it is very important for the project's success to have a common language used between domain experts and developers. Without such language multiple transitions will be necessary. The overall cost of all translations, plus the possibility of misunderstanding, will put the project at risk. Ubiquitous in the team's work, that language should be structured around the domain model. In DDD a ubiquitous language can be materialized as one or more DSLs. These languages are also part of the Software Factories [5], where the process of modeling and implementing software product families realized in such a way that a given system can be automatically generated from a specification written in a domain-specific language.

There are several approaches to exploiting domain-specific languages in development. A DSL program could be interpreted or compiled, or can be used as a model to drive the process of code generation of GPL program chunks or even entire tiers of the developed system [6].

The remainder of this paper is structured as follows: In Section 2 the benefits and drawbacks of domain-specific languages are analyzed. In Section 3 DSLs are observed as specific modeling languages. In Section 4 DSLs (and modeling) and code generation are analyzed in view of the changes of development requirements. The evolution of DSL as a result of the evolution of the software systems is discussed.

2. BACKGROUNDS
A well-designed DSL should be based on the following three principles [7]:

- A DSL provides a direct mapping to the artifacts of the problem domain.
- DSL must use the common vocabulary of the problem domain. The vocabulary becomes the catalyst for better communication between developers and business users (domain experts).
- The DSL must abstract the underlying implementation. The DSL cannot contain accidental complexities that deal with implementation details.

2.1 Pros and Cons
There are many discussions on the web about the advantages and disadvantages of DSL. In fact, the better their design is, the easier the process of writing programs becomes.

Pros
Domain-specific languages have different expressive power compared to the general purpose languages, but they can significantly shorten the time for the development of an application, they can improve the correctness of the developed application, and the communication between the domain expert and the programmer. DSL can be used as mechanism to protect software systems as intellectual property and be a very powerful tool for creating a self-documented code. With DSL multiple programming paradigms can be combined and syntactic noise can be rapidly lowered.

Cons
Regardless of the lower final cost of the overall development a higher starting price of the application development is often pointed out as a disadvantage. Developing application that involves building appropriate DSL is a hard process that requires programmers to be language experts as well. In such

cases the creation of DSL requires complete knowledge of domain constraints. Debugging and unit testing is hard to perform when DSL is used in implementation. DSLs can lead to language cacophony. Proper selection of DSLs and adequate usage is crucial.

2.2 Taxonomy

The availability of language tools aka language workbench is important for the creation and future use of DSL. When can a language be qualified as domain-specific? A common indicator of a DSL is that it is not Turing-complete. These languages can be categorized as external or internal [8].

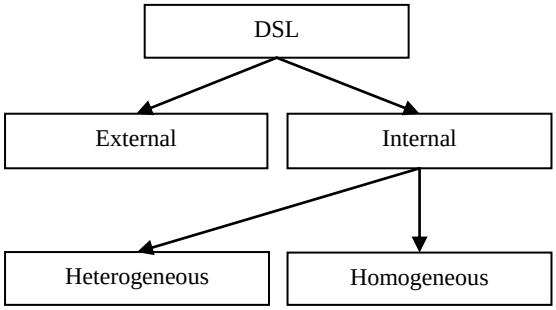

Fig 1: Taxonomy of DSL

An external DSL is a language that is different from the main language (usually GPL). Common examples are languages like SQL, CSS and HTML. Most of them are bound to a particular technology or infrastructure. Often such DSLs are interpreted or translated through code generation tools into GPL code. Ever since XML gained popularity, many external DSLs have been modelled upon it. Some XML based languages were actually equipped with nice graphical outlook. The advantages of external DSLs include: loose specification and minimal or no following of common standards. In such a way developers can express the domain artifacts in compact and useful form. The quality of such a language strongly depends on the ability of the developer to write a high quality code generator or interpreter. External DSLs are not symbolically integrated with the main language and thus things such as refactoring are hard (or even impossible) to be automatically implemented. On the other hand, internal DSL is embedded into the main language and thus it is completely symbolically integrated in it. Internal DSLs are a particular way of using GPL. Internal DSLs provide domain friendly syntactic sugar to the existing API, using underlying programming language constructs. In fact there are two approaches for implementing internal DSLs – heterogeneous and homogeneous. Under the heterogeneous approach, the host language and the embedded language are not processed by the same compiler/interpreter. Two different compilers/interpreters are needed – hence the term heterogeneous. In the case of homogeneous implementation, the host compiler/interpreter is reused or extended so that the host and embedded language are processed by the same compiler/interpreter.

Some general purpose languages are well suited to be extended with internal DSLs. There is an ongoing discussion in the software developers' community on how the quality of internal DSL depends on the features of the host GPL. Martin Fowler and Eric Evans refer to internal DSL as a fluent interface. This term emphasizes the fact that an internal DSL is really just a particular kind of Application programming interface (API), but API designed in such a way that its vocabulary is suitable for sentence-like constructions, rather than sequence of method calls, and the constructions make sense even in a standalone context [9]. Because internal DSLs comply the host language syntax they are not quite readable to non-developers as some of the external DSLs. The grammar of the host language imposes restrictions on the expressive possibilities of the internal DSLs.

Depending on the host language there are different approaches and efforts which need to be developed in order to extend the language with internal DSLs. Some of the host languages are already dynamic unlike others where it could be a challenge to achieve this flexibility.

In summary, the approaches for DSL development could see them as interpreted, compiled, preprocessed, embedded or hybrid, or in the form of fluent interface. In their daily programming tasks software developers often need to choose between command-query API and fluent interface. How should they make a decision whether to build the API as a fluent interface or transform the well-known API into internal DSL? These questions are sometimes answered by developers in their daily tasks by way of creating different helper classes. Developers "carry" these helpers from project to project and they represent their vision how to improve the commonly used API. Sometimes these helper classes are written spontaneously and sometimes deliberately and carefully. In fact depending on the developers' experience, they can turn into proper DSL implementation that will remain stable throughout all similar projects or will be abandoned and completely overwritten in the next project.

3. MODELING WITH DSL

External DSLs are not Computer Aided Software Engineering (CASE) tools. After the rise of CASE during the 90s of the last century, CASE dramatically failed. Martin Fowler [10] summarizes the reason for its failure:

"*I think CASE tools failed for a number of reasons, but underlying it all was the fact that they couldn't come up with a coherent programming environment that would allow people to build general enterprise applications more effectively than the alternatives.*"

After the CASE another continuously evolving approach emerged. The Model Driven Architecture (MDA) was announced in 2001 by Object Management Group (OMG) as a tangible implementation of Model Driven Design based on Unified Modeling Language (UML) which was adopted by OMG in 1997.

In fact DSL appears to be a counterpart to MDA (and UML) approach. Domain-specific languages allow software engineers to focus on design decisions directly related to the particular domain (problem).

DSL programs can be viewed as models and processed by model-driven (metadata-driven) code generators.

3.1 Models and Model Transformations

Three major transformations can be identified, namely: model transformation, model extraction and code generation. Most of the present integrated development environments (IDE) can perform model extraction, for example the generation of class diagrams. Object-relational mapping (ORM) diagram generation from database schema is an example of model transformation. Transformation of ORM diagram to GPL code is an example for code generation.

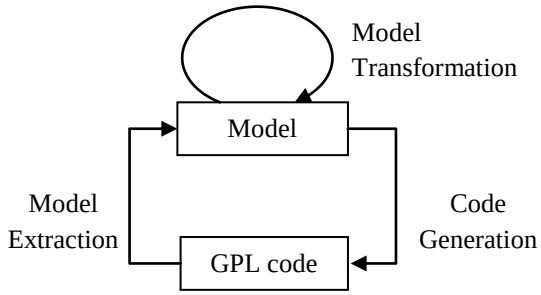

Fig 2: Model transformation and code generation

3.2 Model-Driven Code Generation

Domain-Specific Model (DSM) driven development and transformation from DSM to code require careful design so that they can become really usable. Sometimes partitioning models (partial models) are proposed as improvement to maintainability and understanding. This also adds benefits to model management in multi-user environments [11].

As an effective way to manage the complexity of software development, the modeling provides [12]:

- Better understanding of software systems, and a way to create and communicate software designs before committing additional resources.
- An effective way of traceability through software development process.
- Ability to visualize entire systems and manage complexity.
- Preliminary software correctness through model verification.
- Better cost and time estimation.

Where can code generation techniques be applied effectively? People would argue that code generation should be used as much as possible, but experience shows that there will be a negative effect resulting from covering the whole application. This will dramatically complicate the DSLs used. Complex DSLs are hard to manage, especially when there are requirements changes.

4. HANDLING CHANGES AND DSL EVOLUTION

Requirements management is a very important part of project development as the change in requirements adds to the complexity of the project development. This is also true when DSL is utilized in the development process.

The process of development should be adaptable to:

- Functional changes - such as the inclusion of new functionalities or change the existing ones.
- Non-functional changes: e.g. to change the security, reliability, usability and system performance.
- Changes in the platform - move to new hardware or OS platform.

Regardless of the applied methodology requirements are often not fully provided and programmers identify new requirements or requirements changes in the process of development of software product.

Some DSL are created from scratch just for developing a certain system. On the other hand technology related DSLs remain stable over time and undergo a long process of improvement and standardization. A well-designed DSL should not be affected by requirements changes, but the underlying code generation process and the resulting GPL code usually are. The language developer gains his knowledge for the domain during code writing and this affects the language itself.

Table 1. Influence of requirements changes

	Model/DSL	**Code Generator**
Functional changes	●	○
Non-functional changes		●
Platform changes		●
● – high influence ○ – low influence		

Functional changes mostly affect the written model and if there are no DSL constructs, this can influence the language improvements. The evolution of language will cause changes in the code generator as well. On the other hand non-functional changes affect non-functional aspects weaved in the application. The weaving process is handled by the code generator. Moving a developed product to a different target platform should not affect functionality, business logic or appearance. Platform changes involve non-functional changes, for example different OS introduce different security issues whereas changing hardware can introduce performance issues. In addition, platform changes may advance in switching to different underlying GPL.

Whether you use DSL in single large project for a long time or in many projects for a short time, it will evolve along with the understanding of the problem domain, and it is crucially important for a strategy to be developed on how the domain-specific language should be maintained to mitigate the threat of abandonment at a later stage. The threat level may vary depending on the type of the language (internal or external), target of the language (architecture, technology, or problem domain), as well as the expected features of the language (such as backward compatibility, automatic migration to the new language version of all old programs/models), etc.

Table 2. Influence of requirements changes

DSL		**Abandonment Risk**
Type	External	♦
	Internal Homogeneous	●
	Internal Heterogeneous	■
Target	Architecture	●
	Technology	●

	Problem Domain	♦
Requirements	Backward Compatibility	■
	Automatic Migration	♦
♦ – high, ■ – average, ● – low		

Empirical data on DSLs usage in different software development projects and the cases when DSLs have been abandoned are summarized in Table 2.

External DSL rely on a larger number of tools than internal one. The maintenance of the developed product deteriorates with maintenance of these tools, which is sometimes hard. Moving a project to a new team is often accompanied by misuse or misunderstanding of external tools (even those not connected with particular DSL). Fluent interface, on the other hand, is easy to maintain because the refactoring of the product code and the internal DSL is blended. Requirements, such as backward compatibility and/or adding new tools' functionality for automatic migration to the newer version of domain-specific language, also bring high risk. The highest risk resulting from the abandonment of DSLs, however, is misunderstanding in the problem domain.

5. CONCLUSIONS

Slowly but surely domain-specific languages take their place in software development. After more than 30 years of development they compete with general purpose languages. Software developers turn from single language experts to polyglots. Metaprogramming becomes the main paradigm in programming. The era of command line compilers and MAKE tool is already forgotten - the modern IDE have replaced them. But the course of time keeps changing and the Language Workbenches are approaching.

The immediate future requires solutions that will allow the implementation and use of DSLs on wider scope. It is up to the developer to decide on the way in which a DSL will be implemented and used. Some may choose to implement declarative DSL or imperative, internal or external. Nevertheless, a well-designed DSL should capture the essence of the application domain, and in that sense, there is no better tool to develop the software system.

The successful completion of software project, requires proper understanding of software risks one of which is the risk of abandonment of domain-specific languages during the development process. These languages evolve along with the understanding of the problem domain and product development progress. Keeping them and related tools up-to-date together with proper documentation of their usage will mitigate the risk of abandonment in later stages.

6. REFERENCES

[1] Hudak, P. 1998 Modular domain specific languages and tools. In Proceedings of the 5th International Conference on Software Reuse (JCSR'98), P. Devanbu and J. Poulin, Eds. IEEE Computer Society, 134–142.

[2] Bentley J. 1986 Little languages. CACM, 29(8), 711–721.

[3] Ward, M.P. 1994 Language-oriented programming. Software-Concepts and Tools 15(4), 147-161.

[4] Evans, E. 2003 Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional

[5] Greenfield, J., Short, K., Cook, S., Kent, S. and Crupi, J. 2004 Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley Computer Publishing

[6] Damyanov, I., Holmes, N. 2004 Metadata Driven Code Generation using .NET Framework, Proceedings of International Conference on Computer Systems and Technologies - CompSysTech'2004, 1-6

[7] Debasish, G. 2011 DSLs in Action, Manning Publications

[8] Verna, D. 2013 Extensible Languages. Blurring the Distinction between DSL and GPL, Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, Marjan Mernik Eds, IRMA International, 1–31

[9] Fowler M. 2010 Domain Specific Languages, Addison-Wesley Professional

[10] Fowler M., Blog http://www.martinfowler.com/bliki/ModelDrivenArchitecture.html (visited 05.02.2015)

[11] Warmer, J. B., Kleppe, A. G. 2006 Building a Flexible Software Factory Using Partial Domain Specific Models, In: Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06)

[12] Cemosem, G., Naiburg, E. 2004 The Value of Modelling. A Technical Discussion of Software Modeling, IBM