# Domain-Specific Languages

**Vassilios Karakoidas**
*Department of Management Science and Technology, Athens University of Economics and Business, Athens, Greece*

**Abstract**
Domain-specific languages (DSLs) are programming languages that are by design focused to one domain. The specificity of their syntax and features permits efficient representation of domain concepts expressed within a program, thus enabling the involvement of experts in the software development process. In addition, DSL usage in software development leads to increased productivity for the programmers. This entry provides an overview on DSLs, their design, implementation, and usage.

## INTRODUCTION

Domain-specific languages (DSLs), also known as *micro-languages* or *little languages*, are programming languages designed to focus on a particular domain.[1] Well-known DSLs include regular expressions, markdown, extensible markup language (XML), and structured query language (SQL). General-purpose languages (GPLs) have a wider scope. They provide a set of processing capabilities applicable to different problem domains. Mainstream GPLs are Java, C/C++, Python, and Scala.

To better understand the differences between DSLs and GPLs, consider the following example. The C programming language is a GPL. It provides the basic forms for abstractions and computation. What happens if someone wants to define a matrix of integers in C? An array of pointers must be declared like the following:

```
int **matrix;
```

To access the values of the matrix, the programmer will have to write complex pointer arithmetic statements. If one attempts to implement an algorithm for the multiplication of matrices, a function must be defined that accepts the two matrices as parameters and returns the result.

```
int **multiply(int **m_first, int **m_sec);
```

More advanced languages such as C++ and Java provide more advanced methods to create abstractions; thus there are classes and interfaces. A typical implementation of the matrix multiplication algorithm would have created a class named *Matrix* with a method called *multiply*. For example, in Java, the code would look like the following:

```
public class Matrix {
  public void mupltiply(Matrix m) { ... }
}
```

This approach has many benefits if compared to the C version. The domain abstraction, which is the matrix, is declared directly as a type. In addition, it also contains the method multiply, which is closer to the reality of the mathematical domain.

With modern programming languages, it is easy to create complex libraries that declare and implement the abstractions of specific domains, but there is a barrier; *the syntax of the language must always be used*.

Consider now *octave* or *mathematica*, a language created specifically to deal with this algorithm implementation. These DSLs are used massively for simulations and mathematical modeling. Does anyone consider mathematica's language to develop a web server or a database management system? Those languages focus on the mathematical domain only. Outside it, they have no meaning. The languages of *mathematica* and *octave* are DSLs.

The rest of this entry is structured as follows; first, a brief glimpse on DSL advantages and disadvantages is presented, along with a basic terminology. Three popular DSLs are also presented, with small practical examples of their use. The next section emphasizes on DSL design and implementation patterns. Finally, the entry concludes with the analysis of various works on programming language embeddings and the basic elements on how all these methods can be combined to enhance the overall DSL design and implementation process.

## MOTIVATION

This section enumerates the advantages and the disadvantages of DSL usage for the domain of software development. They are divided into *technical* and *managerial*.

From the *technical* point of view, the usage of DSLs offers overall *performance gains*, in terms of code execution and security. Their usage also encourages *reusability* and *testing* for domain-specific code. However, by using them, the programmers must learn and maintain code in different programming languages, which is a serious disadvantage.

*Encyclopedia of Computer Science and Technology, Second Edition* DOI: 10.1081/E-ECST2-120052315

From a *managerial* perspective, DSL usage can be expected to increase the productivity of the programmers. In addition, using the DSL abstractions permits domain experts, usually not programmers, to help with the design and validation of the system. On the other hand, using DSLs leads to increased development costs in terms of adoption in the software development process, since it is typical for custom tools and libraries to be developed.

## DEFINITIONS AND TERMINOLOGY

First, the term *programming language* should be defined. A formal approach by the IEEE Computer Society[2] proposed a definition of a *programming language* through the definition of *computer language*:

> *(1) A language designed to enable humans to communicate with computers and computer systems and (2) language that is used to control, design, or define a computer or computer program.*

And then the programming language is defined as

> *A computer language used to express computer programs.*

And finally concluding to the definition of the GPL:

> *A programming language that provides a set of processing capabilities applicable to most information processing problems and that can be used on many kinds of computers.*

The term domain-specific languages was provided first through the definition of *application-oriented languages* by the IEEE Standard Glossary for Computer Languages:

> *An application-oriented language is a programming language with facilities or notations applicable primarily to a single application area; for example, a language for computer-assisted instruction or hardware design.*

Finally, Deursen et al.[3] proposed the following definition:

> *A domain-specific language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library.*

## POPULAR DSLS

One of the older DSLs is APT,[4] which was used to program numerically controlled machine tools. It was introduced in 1957. Since then, many DSLs were developed for several domains. In this section, three of the most popular DSLs will be analyzed, namely, regular expressions, SQL, and XML. Table 1 provides a list of the built-in DSL that are supported by the Java software development kit (SDK).

**Table 1** DSL support in Java software development platform

| DSL | Description |
|---|---|
| Regular expressions | Pattern-matching language, implemented in *java.util.regex* |
| SQL | Database query language, implemented in *java.sql* |
| XML | Encoding documents in machine-readable form, implemented in *javax.xml* |
| XPath | Query language for XML documents, implemented in *java.xml.xpath* |
| XSLT | Transformation language for XML documents, implemented in *javax.xml.transform* |
| Oid | Kerberos v5 Identifier, implemented in *org.ietf.jgss.Oid* |
| HTML | HyperText Markup Language, implemented in *javax.swing.text.html* |
| RTF | Rich Text Format, implemented in *javax.swing.text.rtf* |
| URI | Uniform Resource Identifier, implemented in *java.net.URI* |
| Formatter | An interpreter for print-style format strings, implemented in *java.util.Formatter* |

### Regular Expressions

Regular expressions are by far the most popular DSLs. They are used for performing pattern matching and text validation. Their type and syntax variant define a regular expression engine.

- *Non-deterministic finite automaton (NFA)*: The engine represents the regular expressions internally as a non-deterministic finite automaton. NFA-based libraries are slower, but allow syntactically richer regular expressions.
- *Deterministic finite automaton (DFA)*: DFA-based implementations construct a deterministic automaton to represent the regular expression. DFA-based engines are faster and predictable, always returning the longest leftmost match.

Regular expressions come in three dominant syntax variants. These are:

- *Traditional UNIX*: The traditional UNIX regular expression was the first syntax variant that was introduced. It has been replaced by the POSIX standard, although many UNIX applications and utilities, e.g., the command-line utility *sed*, provide support for it.

- *POSIX (extended) modern*: POSIX extended regular expressions to provide an extension to the aforementioned syntax. Three new operators are supported, namely, the union operator ("|"), the zero or one matching operator ("?"), and the one or more matching operator ("+").
- *Perl compatible regular expressions (PCRE)*: The PCRE variant extended the syntax to provide functionality for back referencing, assertions, and conditional subexpressions.

Table 2 lists the regular expression basic set of operators for the traditional UNIX and the POSIX variants.

Regular expressions have built-in support for almost all mainstream languages. The most common syntax variant that is being used is the POSIX. Notably, Perl provides deep integration with regular expressions with the introduction of operators in the base language syntax. It follows the *language extension* implementation pattern, and it is analyzed thoroughly in this section. The following code excerpt exhibits the usage of regular expressions in the Java programming language, by using the built-in application library of its SDK:

```
import java.util.regex.*;
public class SimpleRegex {
    public static void main(String[] args) {
        Pattern pat = Pattern.compile("[0-9]+");
        if (pat.matches("123")) { System.out.
            println("Number"); }
    }
}
```

## Structured Query Language

SQL is the de facto standard language for database querying. Although SQL is defined by an ANSI standard, there are many different dialects of the SQL language, each one usually accompanying each relational database management system (RDBMS). In general terms, SQL offers query execution and data retrieval from a database, creation, update, and deletion of records, definition of the database schema, creation of new databases and tables, and permission definition and update. SQL is a declarative language, and each program consists of executable statements, which are passed and executed one by one in the database.

SQL is supported in many programming languages via a common application programming interface (API), such as JDBC in the case of Java. C, C++, and Ruby do not provide a common API; thus, each database driver implements its own. Table 3 lists the support for SQL by the most popular programming languages.

The following code exhibits SQL usage in Java via the JDBC API:

```
import java.sql.*;
public class SimpleQuery {
    public static void main(String[] args) {
        try {
            Connection c = connectToDB();
            Statement st = c.getStatement();
            st.executeQuery("select * from
              customer");
        } catch(Exception e) { ... }
    }
}
```

## Extensible Markup Language

XML) is a DSL, which is used to define and create documents. XML provides an ecosystem of languages and tools, which drastically eliminate the implementation effort of common tasks, such as searching, validating, and transforming XML documents to other text formats. Table 4 provides a list of the core languages that comprise the XML ecosystem. Each language is a unique DSL and all of them are created and maintained as an open standard by the WWW Consortium (W3C).

**Table 2** Regular expression operators

| Operator | Description | Example | Matching data |
|---|---|---|---|
| c | Single character match | b | b |
| . | Any character match | 1.*2 | 1abc2 |
| [c] | Character group | [a-d]+ | abcd |
| [^c] | Negated group | [^a]+ | bcd |
| c* | Zero or more character match | ba* | b, baaaa |
| $c^+$ | One or more character match | bc+ | bc, bcccc |
| c? | Zero or more character match | bc? | b, bc |
| ^ | Start of line | ^b | b |
| $ | End of line | b$ | b |
| c{x, y} | Bounded occurrences of character c | ab{1,2} | ab, abb |
| r \| c | Match character r or c | b\|d | b, d |

**Table 3** SQL support in mainstream GPLs

| Programming language | Support method |
|---|---|
| Java | Common API (JDBC) |
| C/C++ | External support |
| PHP | Common API (PDO) |
| C# | Common API (ADO.NET) |
| Python | Common API (DB-API) |
| Ruby | External support |
| Scala | Common API (JDBC) |

Since XML focuses on document definition and manipulation, it is used mainly to create an XML family of DSLs, such as XSLT and X-Schema. A typical XML document follows:

```
<?xml version="1.0"?>
<body>
    <item id="1">body of text/item>
</body>
```

All XML documents should have the "xml" tag, which denotes that the following markup file is XML. Tags and attributes are customizable. Programmers may define their own schemas of tags and attributes to describe data and exchange them to other machines. In addition, humans can easily read XML documents.

Almost all GPLs provide standard libraries to support XML. In addition, many of them provide built-in support for XSLT and XPath. C and C++ are perhaps the only popular languages that depend on third-party libraries for XML parsing and manipulation. The following listing exhibits a Python program that opens an XML document and searches for a specific tag:

```
from xml.dom import minidom

doc = minidom.parse(open('customer.xml', 'r'))
node_list = doc.getElementsByTagName ('customer')

for node in node_list:
    print node.nodeName
```

## DESIGN

DSLs are programming languages, and thus are designed by using standard compiler tools and notations. The basic design process of designing and implementing a DSL involves the following phases:

1. *Identify the problem domain*: Initially in the design phase, the primary language entities should be discovered and defined.
2. *Design the semantics*: The semantics of the proposed DSL should be designed. In practice, DSLs are designed through informal processes and many times the semantics are not defined thoroughly.
3. *Design the language syntax*: Define the syntax that will express the core language features.
4. *Implement the application library*: Creation of an application library that implements the DSL's low-level functionality. Usually the application library includes an API for each programming language.

The above design and implementation process provides a straightforward way to implement any DSL, following the *piggyback* implementation pattern. Usually, DSLs are designed and implemented ad hoc, to deal with necessities that rise in complex software development processes. The following section analyses the most common implementation technique as described by Spinellis.[5]

## IMPLEMENTATION AND USAGE

DSL development and integration methods with GPLs are categorized in eight design and implementation patterns.[5,6] The five dominant approaches are *piggyback, language extension, language specialization, source-to-source transformation, and system front-end*.

### Piggyback

Piggyback is the most common implementation pattern. It uses the host language to support the DSL functionality. The DSL can be compiled into the host language or

**Table 4** XML-related technologies and DSLs

| Language | Category | Description |
|---|---|---|
| DTD | Schema definition | Simple language to define and validate XML documents |
| X-Schema | Schema definition | Full-featured XML-based language to define and validate XML documents |
| XSLT | Document transformation | Declarative DSL that provides the means to easily traverse and transform an XML document |
| XPath | Document searching | Simplistic DSL used to query XML documents |
| XML IDs | Searching and validation | Simple mechanism to create IDs for tags and then provide cross-references in a single document |
| XQuery | Document searching | Sophisticated DSL to query XML documents |

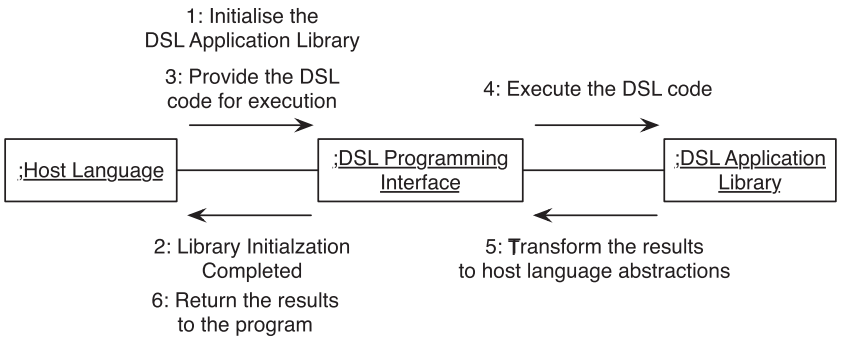


**Fig. 1** The piggyback pattern.

executed by a DSL interpreter. Fig. 1 illustrates the execution process of a DSL using the piggyback pattern. Using the DSL programming interface, which is implemented in the host language, initializes the DSL application library. The DSL code is provided to the application library for execution. The interpreter executes the DSL code and then returns the results to the host language.

## Language Extension

The language extension method extends a host language in order to support a particular DSL. The extension is usually in the form of language syntax.

Perl introduces a series of operators to support *regular expressions*. For instance, the *m//* is the matching operator. The next code excerpt examines by using it if the standard input is a number and prints a message if it matches the number:

```
while(<>) {
    if(m/[0–9]+/){ print "number"; }
}
```

Extending the language with new operators may serve as an adequate solution to support a DSL natively within a GPL, but it is not viable when one needs to integrate many DSLs. With each syntax alteration, the grammar is burdened semantically, making it difficult for the grammar language to evolve smoothly and for the programmer to learn all these syntactic quirks.

## Language Specialization

The language specialization pattern has an opposite effect to the host programming language than the previous one. It specializes the host language with custom support for a specific DSL in order to support it efficiently. Usually, the specialization results in a host language variant is totally incompatible with the original host language.

Powerscript is the core language of the Powerbuilder rapid application development (RAD) tool. The BASIC programming language was specialized to support deep integration with SQL. Powerscript provides syntax and type checking for the integrated SQL queries. The following code excerpt exhibits a parameterized *insert* statement with an integer variable:

```
int loyalty
loyalty = 50
insert into customer (loyalty_points)
 values(:loyalty);
```

## Source-to-Source Transformation

This design pattern assumes that the source code contains the DSL code mixed with the host language code. Then a *preprocessor* analyzes the source code and a *DSL analyzer* transforms the DSL source code to the host language. The host language compiler then translates the generated code into executable form. Fig. 2 illustrates this process.

The *source-to-source transformation* technique differs significantly from the *piggyback* approach since it translates the DSL directly to the host language code. The *piggyback* pattern uses an application library that executes the DSL, and thus depends only on its API.

## System Front-End

The system front-end is the most basic form of DSL implementation. An executable program is developed that provides a front-end to use the DSL. The basic UNIX

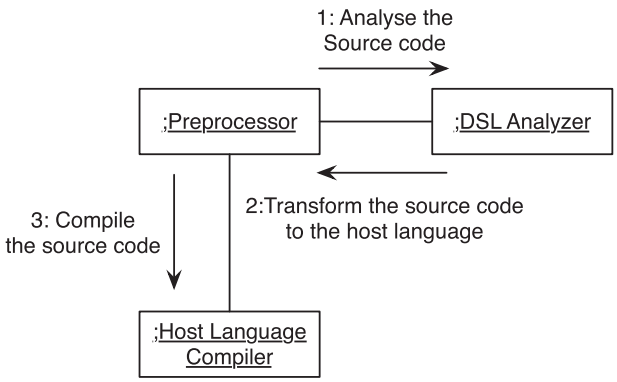


**Fig. 2** The source-to-source transformation pattern.

utilities expose several DSLs, by adopting this method. For example, utilities such as *awk*, *sed*, *tr*, and *grep* are typical examples for this approach. The following command-line example uses the *grep utility*, which provides the command-line facade for the regular expressions DSL, to find all Java source files in a directory hierarchy;

```
$ find . -type f | grep java$
```

## LANGUAGE EMBEDDING AND OTHER APPROACHES

The GPL integration with DSLs brings forth many practical and research issues. An example is the SQL integration with the Java programming language, which is implemented by the JDBC application library. This approach compels the developer to pass the SQL query as a *String* to the application library at runtime. The compiler is completely unaware of the SQL query, and the programmer finds out possible errors usually by a runtime exception raised by the JDBC driver. Such errors usually remain undetected, even with rigorous testing-driven development processes.

The aforementioned design and implementation patterns provide the most common methods to integrate a DSL with a GPL. Mainly, the DSL-oriented research community uses them. Their main focus is to design and develop DSLs and to efficiently integrate them with other programming languages. An example of this approach is the case of regular expressions with Perl. Perl supports regular expressions efficiently and for that it offers a rich set of operators at the language level. This extension cannot be easily adapted for use by other GPLs, without complicating their syntax.

There are other integration techniques that are used by the language designers and the metaprogramming experts. The main concept behind their work is the ad hoc extension of the programming language at compile time, usually by offering complex and exotic mechanisms at syntax level. Their complexity leads to non-practical implementations, which never find their way to mainstream languages such as Java, C#, and Scala.

These approaches also provide DSL embeddings as the by-product of language-mixing mechanisms. In the following sections, several approaches will be analyzed, which partially overlap with the aforementioned design patterns. Those methods deal with the language mixing issue and not with the DSL design and implementation.

### The Metaborg Program Transformation System

Metaborg[7] is an approach based on Stratego/XT and is used for programming language embeddings. With Metaborg, a method is presented that can embed a language within a language at the syntax level. The hosted language is translated through program transformations into the host language, using existing DSL implementations, or by specific code generation targeting optimized performance, security features, and others. In this entry, several case studies are presented, exhibiting various embeddings, even Java within Java. Metaborg method supports type checking between the host and the embedded language.

The following code[8] illustrates a modified Java syntax that introduces inline regular expression support and a match operator (~?), which is used as in the *if* statement. The code is *source-to-source transformed* into plain Java code and compiled by a standard Java compiler. The generated code uses a regular expression library, which is shipped with the standard JDK package.

```
regex ipline = [/
    [0-9]+
    /] ;
if( input ~? ipline )
    System.out.println("An number.");
else
    System.out.println("NOT an number.");
```

### Heterogeneous and Homogeneous Language Mixing

The main issues of DSL embedding relate with the integration and the proper interaction of DSL code with the host language, without having to forfeit the particular domain's established notation. Tratt[9] states that two approaches can be distinguished, namely, *heterogeneous* and *homogeneous* embedded DSLs.

- *Heterogeneous*: Embedded DSLs require a preprocessor to analyze and transform the program code. After that, the compiler of the GPL compiles the code. The preprocessor rewrites DSL expressions into executable GPL code.
- *Homogeneous*: DSLs are application libraries that provide domain-specific constructs. These DSLs are developed in their host language. This method is similar with the aforementioned *piggyback* pattern.

### Pidgnis, Creoles, and Argots

In the European Conference of Object-Oriented Programming 2010, Renggli et al.[10] presented three fundamental types that categorize language embeddings. These are

- *Pidgins*: A *pidgin* extends the semantics of the host language. The embedded language uses the syntax of the host language partially, along with new, but rather limited, constructs.
- *Creoles*: A *creole* introduces a new grammar and then transforms the code to the host language compatible code.
- *Argots*: An *argot* uses the existing host language syntax but changes its semantics.

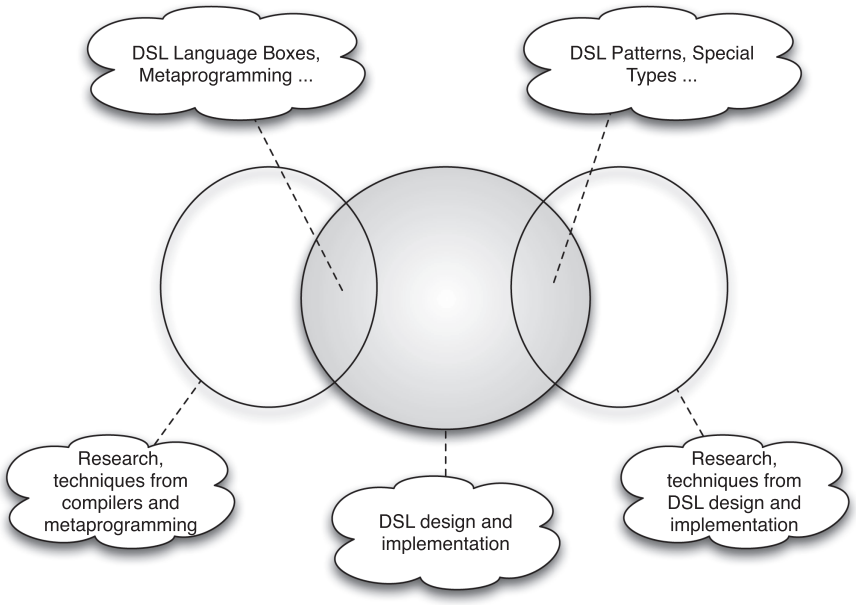


**Fig. 3** Combining DSL approaches with language embedding and metaprogramming systems.

### Language Boxes

Renggli et al.[11] defined *Language Boxes*, a language model that can support embedding of DSLs into a host programming language. They present a case study, which is implemented in Smalltalk and show how regular expressions can be supported efficiently with a small extension of the base language grammar. Practically, the grammar is extended, and the compiler can understand the embedded language through the special notation, and compile it into a host language program, or transform it depending on the needs of the software project.

## FUTURE TRENDS ON DSL DESIGN AND DEVELOPMENT PATTERNS

The future of DSL research lies within both approaches: the DSL language designer should enrich the languages in terms of expressiveness and features and the GPL architects should focus on extensible compilers and adaptable language mixing methods. There must be a fusion of those methods, which for the time being are separate areas of research (Fig. 3). A preliminary fusion of current research indicates that the following three mechanisms should exist in a generic system that features DSL embedding.

- *DSL language code blocks*: A successful approach should provide a way to isolate the DSL code from the host language. The addition of special DSL-oriented operators must be avoided, unless it is crucial to provide refined support for a specific DSL. Language code blocks constitute the perfect trade-off, since they permit native DSL syntax, without breaking language syntax.
- *Special DSL types*: Specialized types should also be defined to aid the compilation process via compile time hooks. These types can give the compiler information that will initiate specific code generation and validation mechanisms at compile time. They should be made possible to be imported arbitrarily and not be a constant burden semantically to the language.
- *Type mapping*: All types of the DSL that will be able to interact with the host language should also be able to be mapped by a type mapping mechanism. The compiler should have information regarding the type compatibility at compile time and raise compiler warnings to prevent errors. Typical type mapping examples exist in the JDBC and ODBC specifications, which map standard Java and C types with SQL ones.

## REFERENCES

1. Ghosh, D. DSL for the uninitiated. ACM Queue, **2011**, *54* (7), 44–50.
2. IEEE Computer Society. IEEE Standard Glossary of Computer Languages (Std 610.13-1993), **1993**.
3. Deursen, A.; Klint, P.; Visser, J. Domain-specific languages: an annotated bibliography. SIGPLAN Notices, **2000**, *35* (6), 26–36.
4. Ross, D. Origins of the APT language for automatically programmed tools. ACM SIGPLAN Notices, **1978**, *13* (8), 61–99.
5. Spinellis, D. Notable design patterns for domain-specific languages. J. Syst. Softw. **2001**, *56* (1), 91–99.

6. Mernik, M.; Heering, J.; Sloane, A. When and how to develop domain-specific languages. ACM Comput. Surv. **2005**, *37* (4), 316–344.
7. Riehl, J. Assimilating Metaborg: embedding language tools in languages. In Proceedings of Generative Programming: Concepts & Experiences, GPCE06, Portland, OR, October 22–26; ACM Press, 2006; 21–28.
8. Bravenboer, M.; Groot, R.; Visser, E. Metaborg in action: examples of domain-specific language embedding and assimilation using stratego/XT. In Proceedings of Generative and Transformational Techniques in Software Engineering International Summer School, GTTSE06, Braga, Portugal, July 4–8, Lämmel, R., Saraiva, J., Visser, J., Eds.; Springer, LNCS, 2006, 297–311.
9. Tratt, L. Domain-specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. **2008**, *30* (6), 1–40.
10. Renggli, L.; Girba, T.; Nierstrasz, O. Embedding languages without breaking tools. In Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP 2010, Maribor, Slovenia, June 21–25, Hondt, T., Ed.; Springer, LNCS, 2010; 380–404.
11. Renggli, L.; Denker, M.; Nierstrasz, O. Language boxes: bending the host language with modular language changes. In Proceedings of Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, October 5–6, Brand, M., Gasevic, D., Gray, J., Eds.; Springer, LNCS, 2009; 274–293.

## BIBLIOGRAPHY

Dubochet, G. *Embedded Domain-Specific Languages Using Libraries and Dynamic Metaprogramming*. PhD Thesis, EPFL, Switzerland, 2011.

Fowler, M. *Domain-Specific Languages*; Addison-Wesley: Upper Saddle River, NJ, 2011.

Ghosh, D. *DSLs in Action*; Manning Publications Co: Greenwich, CT, USA, 2010.

Kutter, P.W. *MONTAGES: Engineering of Computer Languages*. PhD Thesis, Swiss Federal Institute of Technology, Zurich, 2004.

Thibault, S. *Domain-Special Languages. Conception, Implementation and Application*. *PhD Thesis*, University of Rennes, France, 1998.

Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.

Vogt, J.C. *Type Safe Integration of Query Languages into Scala*. PhD Thesis, RWTH Aachen University, Germany, 2011.