Development of Internal Domain-Specific Languages: Design Principles and Design Patterns

Sebastian Günther

A great part of software development challenges can be solved by one universal tool: Abstraction. Developers solve development challenges by using expressions and concepts that abstract from too technical details. One especially supportive tool for abstraction are domain-specific languages (DSLs). DSLs are specifically crafted languages that support abstraction in two ways. On the one side, they abstract from the (real-world) concepts of the domain by introducing objects with structure and behavior that are representative for the domain. On the other side, they abstract from a programming language and do not depend on hardware-specific details. We focus on internal DSLs, which are build on top of an existing host language. Such DSLs can completely use the development tools of their host, are quick to build because they reuse existing abstractions, and can be easily integrated with other programs, libraries, and DSLs of the same host.
This paper's contribution is a set of six DSL design principles and a catalog of DSL design patterns. We differentiate into three types of patterns: Foundation patterns, which provide a DSL's basic objects and operations, notation patterns, which provide the overall layout of the DSL and help to design individual expressions, and abstractions patterns, which help to implement the domain-specific abstractions. Each pattern identifies a development context, design goals that support the design principles, a general solution and a concrete example. To show the applicability of our research results, we document the pattern utilization in 12 other DSLs for the selected host languages Ruby, Python, and Scala. Finally we extend the patterns towards a pattern language by detailing the relationships between the patterns and by showing how other patterns can be incorporated with our patterns.

A Note to the Reader
This paper is a shortened summary of the authors PhD thesis, intended to be self-sufficient. The main focus of the thesis is to provide an extensive amount of syntactic and semantic patterns to develop internal DSLs.
For the workshop, we wish to have a focus on sections 3–5 (around 16 pages) which explain the DSL design principles and three selected syntactic patterns. Sections 2 and 6–7 are illustrating the bigger picture of this work.
We assume our readers to have a basic understanding of what a DSL is. Section 2 can give unfamiliar readers an introduction to this field. Furthermore, to understand the patterns that we suggest, we assume our readers to have at least 1 year programming experience with a dynamic language such as Ruby, Python, or Scala. Helpful, but not required for the patterns explained in this paper, is to understand the metaprogramming concepts of the chosen programming language.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns*

General Terms: Languages

Additional Key Words and Phrases: domain-specific languages, embedded languages, dynamic programming languages, Ruby

Author's address: Sebastian Günther, Software Languages Lab, Faculty of Sciences, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium, email: sebastian.guenther@vub.ac.be

PLoP'11, October 21-23, Portland, Oregon, USA. Copyright 2011 is held by the author(s). ACM XXX-X-XXXX-XXXX-X

1. INTRODUCTION

Domain-specific languages (DSLs) are versatile tools that help to solve one of the biggest challenges in software development: to express development concerns with suitable abstractions. This central *motivation* provides the start of our research. As we will see, there are two forms of DSLs: External DSLs and internal DSLs. In this paper, we focus on internal DSLs: Because an already known host language can be used by the developers, familiar notations and abstractions can be used to design DSLs, and the embedding and absorption of DSL expressions into application code or into each other is simpler. Based on this observation, we shortly define the *knowledge gap* that existing literature and research does not cover. We fill this gap with the contents of this paper, for which we provide an final *outline*.

1.1 Motivation

Domain-specific languages (DSLs) are tailored towards a specific application area [84]. They use appropriate abstractions and notations to represent the domain [85]. Abstractions are the essence why DSLs support software development concerns. Software development means to formulate the concepts of the problem space (domain model) as concepts of the solution space (application components) [27]. In the problem space, domain concepts and their relationships are represented as a domain model. In the solution space, domain concepts are typically represented as objects and methods of a programming language, or other suitable constructs. This facilitates to test, build, and deploy applications. DSLs close the gap between the problem space and the solution space by providing abstractions that bridge both spaces (see ►Figure 1).

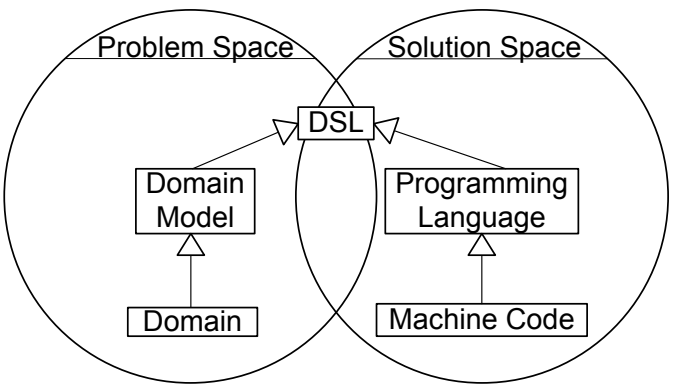

Fig. 1. DSL's are abstractions from the problem space and the solution space simultaneously.

DSLs are credited with effects such as increased productivity, efficient code-reuse, reduction of errors and the focus on the problem space [27][37]. Challenges associated with DSLs are the need to combine domain knowledge and language development expertise, which is hard because few developers have both [60]. Also, language support and standardization are missing [60] and the potential maintenance costs [65] can be very high.

There are two kinds of DSLs: external and internal ones. *External DSLs* require building the language from scratch, with custom syntax and semantics. This comes at the cost of implementing the needed compilers/interpreter and other tools by oneself or with the help of a language development environment. On the contrary, *internal DSLs* are built on top of an existing programming language, also called the host language. This allows to use the complete existing language infrastructure [60], including compiler and interpreter.

In general, DSLs should be developed only if they are expected to have a profound impact on development productivity and are reused in future developments [60]. But in comparing the two types, we see that developing

external DSLs requires much more effort than those of internal DSLs – even if a language development environment is used:

(1) Developers have to learn a new tool that needs to be integrated into the current tool landscape, with all common maintenance burdens.
(2) Developers need to learn at least one formal language to express the syntax and semantics of the DSL. However, most developers have difficulties in applying formal specifications [76].
(3) Since the external DSL gives total freedom, no fixed scheme exists and language creation becomes a strenuous activity that only experienced developers are able to master.
(4) Once finished, the external DSL requires high maintenance. Evolving an external DSL includes modifying its artifacts: surface syntax, the code generator, the generated code and the interaction between the generated code and the application it is used in. Changing one artifact may lead to cascading changes in the others too.
(5) Finally, using the developer's time just to learn a new language can be a waste if they still have much to learn about the main programming language used to implement most of the program artifacts. Better understanding of the main programming language can lead to better productivity.

If developers sacrifice the total freedom of designing an external DSL, but embrace the challenge to implement an internal DSL, then these problems diminish. Internal DSLs are built by modifying or extending the host language's semantics and by exploiting syntactic variations. The effects of using an internal DSL compared to an external DSL are:

(1) The DSL gains better acceptance by the developers and is simpler embodied in application development, because it is based on a known host language and uses abstractions familiar to the developers.
(2) The host language provides a familiar frame for designing DSLs. Embracing the restricted syntax and semantics of the host language is a good way to promote language design creativity.
(3) No new tools have to be learned or added. Quite the contrary, because DSL expressions are also expressions of the host language, basic IDE support is given. Extensible IDEs, such as Eclipse, also allow custom syntax highlighting and code completion. This can be exploited to facilitate programming with the DSL.
(4) Using and especially designing DSLs can help the developer to better understand the host language. Understanding the type system, semantics, and metaprogramming capabilities of a language certainly improve the developer's skill and productivity for non-DSL usage too.
(5) Finally, because the DSLs are all based on the same host language, the embedding and absorption of DSL expressions into application code or into each other is simpler. This allows building multi-DSL applications which profit from the effective collaboration of several DSLs [41].

In summary, although internal DSLs are restricted in their syntax and semantics, they provide the following benefits: Less expensive to build, more likely to be accepted, and easier to incorporate into proven development techniques, development processes and tools. For these reasons, the dissertation focuses on internal DSLs. If not explicitly mentioned otherwise, we refer to internal DSLs when we speak of DSLs.

1.2 Knowledge Gap

Although internal DSLs can provide several remedies for todays challenges in software development, there are several knowledge gaps that need to be overcome. Despite the question how to develop them with a process, which we do not tackle here in order to provide a clearer focus, the two main gaps are the DSL design principles and the DSL design patterns.

– *DSL design principles*: DSLs have unique properties that are difficult to express as principles for guiding the DSL development. Most internal and external DSL case studies such as [38][81][9][1][28][11][50] do not discuss design principles or explain their DSL's design properties. Existing work on principles uses different names and explanations for similar concepts. For example, there are principles about designing notations for

visual constructs [23][63], for building abstractions [53], for programming language design [62], or specifically for DSLs [67]. We propose a set of principles that are originating from the definition of DSLs, and we analyze and compress the principles mentioned in other works. These principles form a coherent vocabulary to discuss the design of DSLs.
– *DSL design patterns*: Finally the question remains how to actually implement the DSL on top of an existing programming language. Most case studies do not generalize their approach and mix host language characteristics, as well as language constructs, metaprogramming and design considerations. The case studies suggest to use host language specific metaprogramming methods [26], internal interpreter objects [28][75], object-oriented programming constructs, in particular methods [18][1], or even the host's metaobject protocol for implementing a DSL [28][24]. We provide a thorough analysis and description of the different mechanisms in the form of patterns. Our patterns record the context, problem, and solution of DSL design challenges and are used in DSLs based on Ruby, Python, and Scala.

The patterns are the essential contribution in this work. Patterns help to understand recurring design problems and their solutions. Moreover, as we see soon, they are essential to communicate DSL solutions independent of the host language, because existing case studies talk of very language specific techniques and mechanisms they use, which are hard to take to another implementation or host language.

1.3 Outline

This paper is structured as follows. In ►Chapter 2, we provide additional material on the definition of domain-specific languages. ►Chapter 3 includes our first contribution: the design principles. Following this, ►Chapter 4 explains how each pattern is defined and gives a short outline to all patterns. In ►Chapter 5, we explain the patterns that were selected for this paper: The notation patterns BLOCK SCOPE, METHOD CHAINING, and KEYWORD ARGUMENTS. To show the applicability and the actual usage of our patterns, ►Chapter 6 shows several DSL case studies and details where and how the patterns are used. ►Chapter 7 shows how to evolve the patterns to a pattern language by giving rich information about the relationships of the patterns to each other and by presenting a list of design questions that developers can follow to design a DSL. Finally, ►Chapter 8 summarizes our paper.

We use the following typeset to distinguish difference concepts: *keyword*, `source code`, PATTERN, and *subpattern*.

2. DOMAIN-SPECIFIC LANGUAGES

DSLs are versatile tools that are effective to cope with today's development challenges. This chapter explores DSLs in more detail. We start to give the *historic perspective and definition*, where we see that the idea to provide custom, specialized languages was and is an ongoing concern in software engineering research. Based on the large amount of research, several *DSL properties and types* need to be distinguished. We provide a morphological scheme that lists the most important properties and their individual characteristics. Finally, we complete the exploration of DSL utilization by discussing the *role of DSLs in software development approaches*, such as model-driven development and generative programming.

2.1 Historic Perspective and Definition

The term domain-specific language can be seen from very different perspectives. We think that each perspective is reflected in the different concepts that contribute to today's understanding of DSL. In a period of about 40 years, the following concepts were introduced in the literature:

– *Specific Language*: As early as 1966, the idea to separate the appearance of programs from the underlying abstract entities was introduced. This concept facilitates the provision of a family of languages with individual

members called specific languages. The family is described by similar logical structures and categories, while the specific languages have a physical representation that is tailored towards it's application area [56].
– *Domain Language* – In correspondence with the discovery of the term domain and it's fundamental influence on the application development, the term domain language was introduced. By analyzing the domain, developers derive a domain model that is expressed using a domain language. This language has a declarative character and uses the entities and actions of the domain as words. With the help of the domain language, system components are expressed declaratively and in later development steps made executable by transforming the language into another format [66].
– *Little Language*: These languages point out the importance of a distinguished representation for specific problems. Little languages are focused on the user, they utilize expressions that are clearly expressing the intent of expressions. Such languages are often implemented with suitable preprocessors in order to reduce the effort of writing complex compilers. Therefore, little languages are typically transformed to an intermediate form that is the input to another tool or that is used to generate an executable form [12].
– *Domain Specific Language*: Finally around 1998, the aspects of forming specific representations for a domain and implementing them as languages merged to the term domain specific language. These languages are defined as a "programming language tailored for a particular application domain" [52]. Through appropriate notations and abstractions, DSLs are directed at their domain [85][51][6] and have a high expressiveness [85][81].

By following all above consideration, and especially emphasizing the importance of domains, we define DSLs as follows:

> A domain-specific language is an artificially created language that, through suitable abstractions and notations, embodies syntax and semantic that represents the concepts, attributes, operations, and relationships of a domain as an interpreted or compiled computer language.

Because the DSL is a representation of the domain and executable as well, we also say that a DSL is an executable domain model. Based on this definition, we now explain the different types of DSLs.

2.2 DSL Properties and Types

The long history of DSLs leads to a high number of DSLs that can be identified. We studied their properties, and found five defining properties: *appearance*, *origin*, *originality*, *implementation*, and *purpose*. These properties, and their characteristics, form a morphological scheme (►Figure 2). Each property is detailed in the following subsections.

2.3 Appearance

A DSL's appearance determines its surface syntax. Graphical DSLs consist of abstract symbols and drawings to express the relationship between domain concepts. Textual DSLs use mostly literal characters and mathematical symbols to express their meaning [20].

2.4 Origin

The origin of a DSL determines if the language is developed independent from other languages or whether it is based on top of another language. External DSLs require that their interpreter or compiler is written specifically for this language. This means to separately develop the basic syntax, the expressions and tokens, and their semantics. Internal DSLs use an existing host language to build their abstractions and notations. They can reuse the complete host language infrastructure, but are bound to the syntactic and semantic constraints of the host.

Property	Characteristic				
Appearance	Graphical			Textual	
Origin	External			Internal	
Originality	Original	API Wrapper		Sibling	Mimic
Implementation	Extensible Compiler/Interpreter	Generator	Macro	Template	Metaprogramming
Purpose	Horizontal Domain		Vertical Domain		Development Concerns

Fig. 2. Morphologic scheme for identifying DSL types.

2.5 Originality

Originality is the degree to which a DSL uses an existing language or library. At the one hand, it can be a complete original, and on the other hand, it can mimic an existing language.

– *Original*: An original language encapsulates a domain that is not represented yet by another language or by a library.
– *API Wrapper*: In software engineering, libraries are written in one concrete programming language and can be reused in accordance with their application programming interface (API). This fixed form can be wrapped by a DSL by combining the methods of the API to a language that is more versatile in its application. This kind of DSL is also called fluent API or liquid API [17].
– *Sibling*: The DSL copies parts of the syntax and semantics of an existing language.
– *Mimic*: The DSL mimics the concepts and expressions of an existing language.

2.6 Implementation

DSL literature provides a pile of different implementation techniques [27][85][88][78][60][26][28]. We consolidate these techniques[1] into the following five items ordered according to the required implementation effort.

– *(Extensible) Compiler/Interpreter*: Compiler and interpreter provide full support for source-to-source transformation. They check the input for several conditions, and either generate machine executable artifacts or create an in-memory representation of the program. For an external DSL, the compiler/interpreter has to be written. In the case of an internal DSL, the existing compiler/interpreter can be extended. In this case, especially the implementation of an interpreter can be modified or in the case of a high-level language like Smalltalk, the internal representation of a program in the form of the abstract syntax tree can be modified [70].
– *Generator*: A generator is a sophisticated tool that maps an arbitrary input stream to an arbitrary output stream [27]. Generators are implemented external to the programming language and typically use abstract

[1] We do not explicitly cover two-level languages [27] or multi-level languages [36]. The basic concept of such languages is that the compiler of an earlier stage produces the compiler for the next stage, thereby changing the semantics of statements. This allows optimizing the code with respect to the domain. Each level could use one or several of the presented mechanisms to perform this compiler generation for the next level.

representations of the components. In contrast to compilers and interpreter, generators just produce the code but do not execute them.
– *Template*: Templates[2] are used to represent a computation in a form that is independent of the concrete implementation. Before a template is executed, it needs to be transformed in a target language. There are two options for this transformation. First, using a *preprocessor* that is external to the target language and which can be realized, for example, by utilizing string processing tools of an existing language [8] or parser combinators [75]. Second, a program can implement the quasi-preprocessing of code as an object that actively transforms the template into the internal format or interprets and reacts according to the input – this technique is called *internal interpreter*.
– *Macro*: Macros are similar to templates, a technique that facilitates the separation of the physical representation of some source code from its internal form. The difference, however, is that the semantics and transformation of macros is always defined as a part of the programming language they are defined in. In Lisp for example, macros are a set of compiler directions and host language syntax [72]. Macros are often used to extend the host language with new constructs [10], which can be used to build DSLs [83].
– *Metaprogramming*: The general definition of metaprogramming is that of code that writes code [27]. However, this broad meaning would encompass all above mentioned tools. In the context of this dissertation, we speak of metaprogramming as the capabilities of a programming language. With this narrowing, metaprogramming supports two tasks: *Introspection*, which is the analysis of a program's internal structure, for example its classes and methods, and *intercession*, which is the capability to modify the given behavior [19]. In statically typed languages like C++, metaprogramming can be done via C++ templates[3] for defining DSLs [27]. In dynamically typed languages like Ruby and Python, metaprogramming even allows to change the behavior of built-in objects and methods. Metaprogramming aims at "the creation of new abstractions that are integrated into the host language" [82].

►Figure 3 orders the implementation mechanisms along two dimensions: horizontally into whether they support external or internal DSLs, and vertically according to the abstraction level. While modifying the compiler or interpreter involves understanding the complete transformation process, macros, templates, and metaprogramming are more abstract, require less knowledge to use, and thus facilitate DSL development. Metaprogramming is especially powerful, as it can be used as a strategic tool for software composition and evolution [59], because of its high abstraction level. A special type of metaprogramming is to introspect and modify metaobjects that represent classes, functions, and instances of a language. The sum of all functions that access metaobjects is called the metaobject protocol [54], which is the mechanism with the highest abstraction level to modify the behavior of a language. In order to reduce the effort in designing and implementing DSLs and to build upon existing knowledge of developers, we focus our research on metaprogramming mechanisms.

2.7 Purpose

We identify three purposes of a DSL: supporting the implementation of a horizontal domain, a vertical domain, or a development concern.

Domains can be distinguished into horizontal and vertical domains [27]. *Vertical domains* are business areas like aviation control, portfolio management, and order processing system. *Horizontal domains* are system parts or components like the GUI, database, and workflow system. Both domains can have arbitrary subdomains. The horizontal parts can be detailed, such as different database adapters, database schemes, or components like transaction management and more. Vertical domains can be further divided, for example the aviation control into subsystems that track the movement of air cargo, touristic flights, and military vehicles. A DSL can also

[2] Here, templates are meant in a generic form and not the specific form of C++ templates. In C++, templates are used as class or function template to abstract concrete type information until the code is bound and compiled for a specific type [27].

[3] Which is called template metaprogramming [27], originally shown for C++, but also usable in Haskell [73].

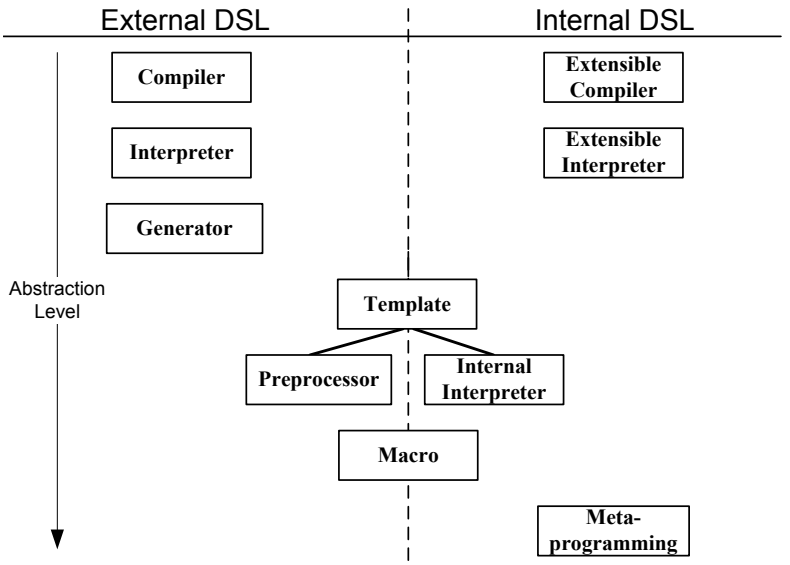

Fig. 3. DSL implementation mechanisms.

support general *development concerns*. Considering the development phases, we can use DSLs to support the specification, testing, and deployment of applications. Considering programming paradigms, some DSLs extend the available paradigms that can be used to structure a program. The following lists give some examples for the different DSL purposes.

Vertical Domains

- Financial products [4]
- Video processing [81]
- Healthcare systems [64]
- Telephone services [57]
- Magnet tests for the Large Hadron Collider at CERN [5]
- IT infrastructure management [45]

Horizontal Domains

- Structure of webpages with HTML
  - Ruby: HAML[4] and Markaby[5]
  - Python: DAML [6]
- Layout of webpages with CSS
  - Ruby: SASS[7] and LESS[8]
  - Python: CleverCSS[9]
- Database schema and queries with SQL-like language

[4] http://haml-lang.com
[5] http://markaby.rubyforge.org/
[6] http://github.com/dasacc22/DAML
[7] http://sass-lang.org
[8] http://lesscss.org
[9] http://pypi.python.org/pypi/CleverCSS/

– Ruby: Active Record[10] and DataMapper[11]
– Scala: ScalaQL [77]

Development Support

– Test/Unit (Ruby, test-driven development)[12]
– Unittest (Python, test-driven development)[13]
– RSpec (Ruby, behavior-driven development)[14]
– Capistrano (Ruby, web application deployment)[15]
– rbFeatures (Ruby, feature-oriented programming) [47][46]
– Aquarium (Ruby, aspect-oriented programming)[16]
– Context-Py (Python, context-oriented programming) [3]
– Actor (Scala, built-in, concurrent programming) [48]
– Python-Constraint (Python, constraint-oriented programming)[17]

2.8 Role of DSLs in Software Development Approaches

One goal of DSLs is to increase the developer productivity and the reusability of source code. These goals are also shared by the following approaches: language-oriented programming, model-driven development, generative programming, and software factories. Each of the approaches is discussed in the following paragraphs, and we explain the (potential) role of DSLs in each approach.

Language-Oriented Programming

Language-oriented programming (LOP) is more of a paradigm than a concrete software development method, which emphasizes the idea to use a custom language for supporting software development or expressing a particular application domain [29]. This makes the connection between LOP and DSL obvious: In order for LOP to express the software in “terms of concepts and notifications of the problem”, DSLs are created and used to develop the application [86][29]. Using DSLs, which are understood here as “domain oriented, very high level [...] languages” [86], leads to better separation of concerns, higher development productivity, and maintainable designs because of fewer lines of code [86].

Model-Driven Development

Representing all concerns and requirements as common source code can get very complex for bigger applications, with possible negative impact on maintenance and evolution. Model-driven development (MDD) engages this problem by providing novel representations that describe and generate software with abstract models [79]. MDD helps to increase the development velocity and quality, to reduce redundant code and to better cope with changes and reusability [79]. Models are not only used for representation, but also for automatizing the build and deployment process of applications [37].

MDD is a transformation-based software development approach that uses graphical and textual models [79]. Graphical models can be based on UML [32], where stereotypes and annotations provide information that is rich

[10] http://rspec.info
[11] http://datamapper.org
[12] http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/
[13] http://docs.python.org/library/unittest.html
[14] http://rspec.info
[15] http://github.com/capistrano/
[16] http://aquarium.rubyforge.org/
[17] http://labix.org/python-constraint

enough to generate concrete source code. For textual models, external DSLs are typically used. A MDD project can combine both graphical and textual models which are ultimately transformed into source code.

Internal DSLs play a neglecting role in MDD: only some case studies report on using internal DSL for the transformation process, such as [25].

Generative Programming

Generative programming is a holistic software development approach that aims to provide and structure a large asset base to form a product line. Three concepts are fundamental in this approach. The *problem space* describes the real-world domain that consists of domain-specific concepts and features. The *solution space* is the domain of computer components and other artifacts that implement a program. In developing software, the problem space is mapped to the solution space by using *configuration knowledge*. This knowledge is based on the structure of the components and how their combination, construction, configuration and optimization can be used to implement the program specified in the solution space [27].

Generative programming is concerned with providing generative components that are accessible for various configuration options. Implementing these components by using DSLs is a viable option. The other use-case for DSLs mentioned in [27] is to provide the mapping of a problem space specification to the solution space.

Software Factories

Software Factories is an approach that strives for a high amount of automization and low costs in creating software [35]. A software factory is defined as a "software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components" [37].

Assembling a software factory and producing software requires three steps. At first, various software artifacts are gathered, similar to the collection of assets for a software product line. Then, a factory schema is developed that completely maps requirements and functionality to the software artifacts. Finally, a template for the factory is created and configured for producing a concrete application [37].

DSLs are supportive of the software factories approach: They can be used to express software artifacts, or used to express the requirements to artifacts mapping in the factory schema.

2.9 Summary

This chapter explained important background information about DSLs. We started with a historic perspective where we identified four different conceptual understandings of DSLs. Each concept contributes to our understanding: A DSL is a compiled or interpreted language that represents the domain concepts with suitable abstractions. Over the course of time, various DSL properties were developed, which we compressed into five defining properties: appearance, origin, originality, implementation, and purpose. With the morphological scheme that we introduced, a DSL can be identified along these characteristics. Finally, we explained the role of DSL's in other software development approaches, where we saw that they are especially supportive for generative programming and software factories. Now, the next chapter details the research need for internal DSLs.

3. DESIGN PRINCIPLES

Design principles provide a structured approach to develop DSLs in a specific manner. In the chapter, we present the following six principles: *Abstraction*, *Generalization*, *Optimization*, and *Notation*, *Absorption*, *Compression*. Although the principles have no fixed order per se, they can be regarded like this: The DSL's syntax is developed to express domain-specific *abstractions* with suitable *notations*. First, the necessary concepts are *generalized* out of similar concepts in the domain, and domain-specific *optimization* are applied for their behavior. Then, the resulting

language can be further *compressed* and obvious expressions *absorbed* into the DSL. Following paragraphs further explain the principles.

Abstraction

Goal: Implement domain concepts as abstractions to deemphasize technical details.

DSLs abstract twofold: From the domain and from their host language. Abstracting from the domain is to include only those entities and operations in the language that are relevant. This means to provide “crisply defined conceptual boundaries” [13] to find entities that “fit the domain closely” [6]. Abstractions from the host language means to combine common, recurring patterns [16][30] in one form. We find abstractions for DSLs at the base level and at the meta-level. On the base level, the host’s basic operational model âĂŞ- expressed by the existing language constructs – is extended with novel expressions. At the meta-level, the existing construct’s behavior is modified or extended. Both forms of abstractions are fundamental for DSLs since they have to step away from the language in order to step towards the domain.

Generalization

Goal: Reduce the amount of concepts by replacing a group of more specific cases with a common case.

Generalization is an important concept in domain engineering and in software engineering. First, the domain is captured in its full scope consisting of multiple concepts and their relationships. They naturally form a hierarchy in which one concept is more general than other specific concepts. Finding such hierarchies helps the developer to understand the domain. Generalization keeps DSLs small as the reduction of the amount of concepts results in a small amount of needed constructs [21]. For the DSL’s syntax, this means to design the language around common concepts. For the DSL’s semantic, the developer then chooses how to prune, reorganize, and combine the concepts in order to express generic and specific cases, using mechanisms such as object-oriented programming.

Optimization

Goal: Implement algorithmic optimizations to improve the DSL’s computational performance.

Producing efficient code is an essential characteristic of programming languages. According to [67], DSLs allow two forms of optimization. The first form is the implementation level, for example optimizing the memory allocation. This optimization is included in the DSL that uses the host language. Then, on top of the implementation, domain optimizations can be implemented by choosing suitable abstractions [67]. This means to choose efficient algorithms and data structures [27]. Optimization improves the computational performance by using suitable models and domain-knowledge with an efficient implementation [15].

Notation

Goal: Represent domain-specific concepts with suitable, clear distinguished entities.

A DSL’s syntax is the sum of the host’s syntax and the domain-specific notations. The syntax consists of all language constructs, keywords, structures, and the rules governing their combination. The rules determine the general layout of the language. Thereby, a “trade-off between naturalness of expression and ease of implementation” [12] has to be made. Internal DSLs cannot leave the syntactic frontiers of their host language, but they can chose to combine existing syntactic modifications to achieve a form that has little or no resemblance to the host language. Providing the appropriate notations [85] through defining clear distinguishable and representative entities defines the DSL’s syntax.

Compression

Goal: Remove tokens and keywords that have no meaning in the domain.

The goal of compression is to provide a concise language that is sufficiently verbose for the domain experts. By reducing the amount of expressions or by simplifying their appearance while the semantics are not changed, the

code footprint of DSL programs is reduced. This leads to better understanding the application language and the domain [55][87].

Absorption

Goal: Absorb domain commonalities into the DSL expressions to facilitate DSL usage.

Characteristic properties of domains are commonalities for a broad set of expressions. Users are supported effectively if these commonalities are absorbed by the language, which means that certain assumptions and concerns are not required to be explicitly expressed. DSLs with high absorption support the user by capturing these assumptions implicitly, relieving him of the burden to repeat himself. This is in stark contrast to a traditional library, where functions are dependent on each other, require a manual setup of the context, and have a restricted call order. The best example for an high amount of absorption are declarative DSLs that just describe a desired state, and the DSL's implementation uses all explicitly and implicitly given information to effectuate this desired state. Absorption provides DSLs implicit focused expressive power [80] by implicitly integrating domain commonalities in the DSL.

4. PATTERN STRUCTURE

In this chapter, we start with explaining the *pattern structure* with which each pattern is presented in the following chapters. This structure helps readers to quickly learn the patterns, where one important part is to show the patterns applicability in the context of one of the *common examples*. To let readers quickly browse through the patterns, we conclude this chapter with a *pattern overview*.

4.1 Pattern Structure

The pattern explanation has a structure that is close to the initial introduction of patterns [34] and recent publications [39; 22; 89]. To show the applicability of the patterns, we research how they can be utilized in three different host languages. We select the two dynamic host languages Ruby and Python because of their popularity as listed with the Tiobe language popularity index [18] and the large number of DSL projects as listed with Github[19]. As a contrast, we also use the compiled programming language Scala which is not so popular on Tiobe, but has several DSL projects listed on Github, and even includes a DSL in its core contribution.

The following structure is used to explain each pattern:

– *Name*: The name of the pattern.
– *Also Known As*: Some of our patterns use an existing foreign pattern to put it into the context of domain-specific languages. Such patterns are introduced with "Also known as". Foreign patterns that are similar to one of our patterns that we published first and earlier in our works [40][43][44] are introduced with "Also defined in".
– *Context*: Explains a particular design or implementation scenario in which the application of the pattern is beneficial (note that the context follows right after the name, without an additional heading).
– *Problem*: A question that raises a common challenge in DSLs, and an explanation of how the challenge negatively effects the DSL.
– *Design Goals*: Explains explicit design goals and DSL design principles that are achieved by applying this pattern. Usually, one pattern can be used to support different principles. In some cases, one pattern can be used in two distinctive ways to support a principle, which we document both.
– *Solution*: A general advice what to do, followed by a source code example illustrating the patterns application, and than a detailed explanation of the solution and its host-language specific implications.

[18] http://tionbe.com
[19] http://github.com

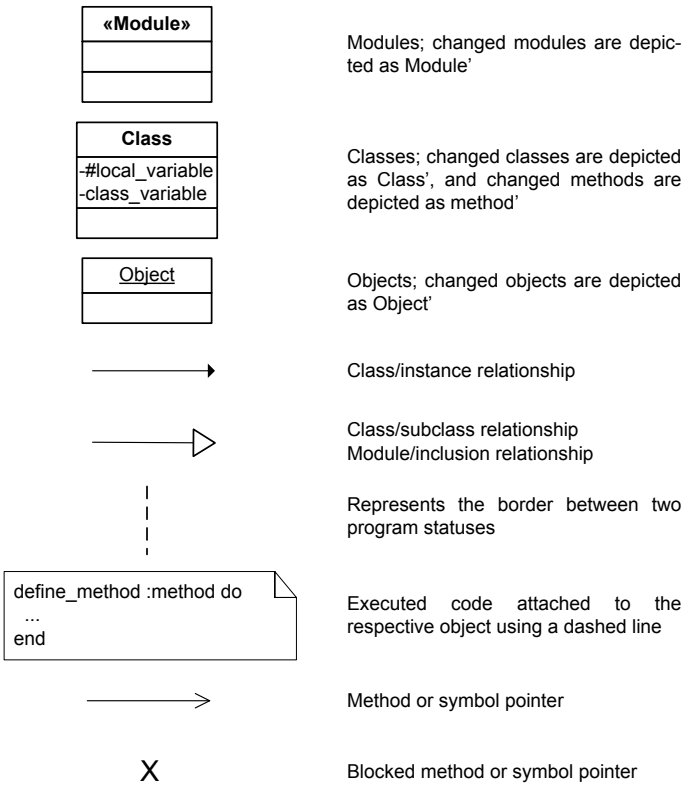

Fig. 4. Symbols used to explain a pattern's solution.

– *Example*: Based on the common examples that are explained shortly, a detailed utilization of the pattern using the Ruby programming language is given. Source code examples and sometimes diagrams provide further illustration (see ►Figure 4 for the used symbols).
– *Liabilities*: Explains potential pitfalls and tradeoffs when this pattern is applied.
– *Known Uses*: Explanation of a pattern's occurrence and the particular meaning it has in the context of a DSL. We list at least one occurrence per host language, as long as the pattern is available in the language (we do not document all pattern occurrences for all DSLs). We choose to analyze the following DSLs:

– *Ruby*

– *Rails* (version 2.3.5) – Web-application framework with a strong Model-View-Controller architecture that provides many convenient expressions as a DSL for web applications. Rails is used in several open-source and commercial projects. This version of Rails is closely bundled with other libraries that we also analyzed: *ActiveSupport* (version 2.3.5) adds helper classes, *Actionpack* (version 2.3.5) provides request and response objects, and *Builder* (version 2.1.2) is used to build configuration objects representing XML-like data structures (http://rubyonrails.com).
– *ActiveRecord* (version 2.3.5) – Provides database abstraction with the philosophy that an object encapsulates the data of one specific database row. Also it can be used as a standalone DSL, it is often bundled with Rails (http://ar.rubyonrails.org/).
– *RSpec* (version 1.3.0) – Framework and DSL for behavior-driven development (http://rspec.info).
– *Sinatra* (version 0.9.4) – Lightweight web framework and DSL that uses declarative expressions for specifying request handling (http://sinatrarb.com).

– *Python*
  – *Bottle* (version 0.8.3) – Similar to Sinatra, a lightweight web framework that uses normal Python function declarations and annotations to specify how queries are responded to in the application (http://bottle.paws.de/).
  – *Should DSL* (version 1.0[20]) – Supports simple behavior-driven testing by adding expressions like `should`, `be`, and `be_greater_then` as expressions in Python (http://github.com/hugobr/should-dsl).
  – *Cryptlang*[21]: A DSL for defining cryptographic algorithms and random number generators [1].
  – *LCS DSL*[21] – The Constellation Launch Control System DSL, or short LCS DSL, provides a command like language for "programming test, checkout, and launch processing applications for flight and ground systems" [11].
– *Scala*
  – *Actors* (version 2.7.7final) – Actors is a built-in Scala DSL for asynchronous messaging, and frequently deployed for concurrent programming. It uses concepts like a messaging box and special operators to send and retrieve messages (http://www.scala-lang.org/api/current/scala/actors/package.html).
  – *Specs* (version 1.6.2.1) – DSL for behavior driven development similar to the block-like nature of RSpec (http://code.google.com/p/specs/).
  – *Apache Camel* (version 1.6.4) – Helps to express the configuration of enterprise systems that consist of Java-compliant applications, allowing to define routes and APIs for the components (http://camel.apache.org/scala-dsl.html).
  – *ScalaQL*[22]: Implementation of SQL as a Scala DSL [77].
– *Related Pattern*: Finally, a list of closely related patterns.

4.2 Common Example

The common example used to illustrate the application of the patterns is the Infrastructure Management DSL. Infrastructure management takes care of the initialization, configuration, and maintenance of several servers forming the infrastructure of a customer. Because of continuously increasing requirements with regard to flexibility, high automatization support that limits the amount of manual involvement becomes crucial. This tasks is supported by a set of three DSLs. We use our developed Machine Description Language (MDL) and the PCML (Package Configuration Management Language) from [49] for examples. ►Figure 5 shows an example for the MDL and the PCML, with the following meaning:

– *Line 1–15*: An example of the MDL that opens a constructor, sets the default name, and passes other configuration options in the form of a block.
  – *Line 2–4*: Configures the type, owner, and the operating system of this machine.
  – *Line 5–11*: A nested block that specifies the hypervisor-specific options, like the Amazon[23] machine image (ami) id and its source, the size, the security group (the security group determines the available services and the open ports of the machine, such as SSH on port 22), and the private key used to access this machine via SSH.
  – *Line 12*: The hostname of the machine which is used inside the network.
  – *Line 13*: Expresses to monitor the CPU and RAM usage of this machine.

[20] The version 1.0 was submitted on June 17th, 2009, we use the more recent commit with the id 1725b62ad30195f578a4, issued on July 15th, 2010.

[21] For these DSLs, we do not have access to the source code. While we can analyze the expression for notation patterns, foundation patterns and abstraction patterns can only be checked in a limited way.

[22] For this DSL, we do not have access to the source code. While we can analyze the expression for notation patterns, foundation patterns and abstraction patterns can only be checked in a limited way.

[23] http://aws.amazon.com

```
1  app_server = Machine "Application Server" do
2    type :ec2
3    owner "sebastian.guenther@ovgu.de"
4    os :debian
5    hypervisor do
6      ami "ami-dcf615b5"
7      source "alestic/debian-5.0-lenny-base-2009..."
8      size :m1_small
9      securitygroup "default"
10     private_key "ec2-us-east"
11   end
12   hostname "admantium.com"
13   monitor :cpu, :ram
14   bootstrap!
15 end
16
17 deploy "Redmine" do
18   enroll "Redmine/Database", :on => app_server + aux_server do
19     #...
20   end
21 end
```

Fig. 5. Example of the MDL and the PCML for configuring a server and for installing software package on it.

  - *Line 14*: Finally an expression to install basic software.
- *Line 17–21*: An example of the PCML to deploy the package Redmine[24].
  - *Line 18*: Redmine has several dependencies to satisfy, and in this case the database is configured to be enrolled on the application server as well as an auxiliary server for obtaining a master-slave relationship. Other packages would be implemented likewise.

4.3 Pattern Overview

We differentiate the patterns into foundation patterns, notation patterns, and abstraction patterns. ►Figure 6 gives a graphical overview to the patterns.

To give a complete overview to all patterns, we explain the problem of each pattern in the following list.

4.3.1 *Foundation Patterns*

Foundation patterns are the very first pattern to apply, they explain how the domain concepts, attributes, and operations are implemented as entities.

- DOMAIN OBJECTS: How to implement suitable abstractions of the domain entities and their relationships?
- DOMAIN OPERATIONS: How to implement suitable representations of the domain operations?

4.3.2 *Notation Patterns*

The notation patterns structure the host's syntactic capabilities to form expressions that provide the domain-specific notations. They are structured into three groups.

- *Layout Patterns*: Provide the general layout of the DSLs, either as vertical blocks or complex, horizontal expressions.

[24] An open-source project management application, see http://www.redmine.org/

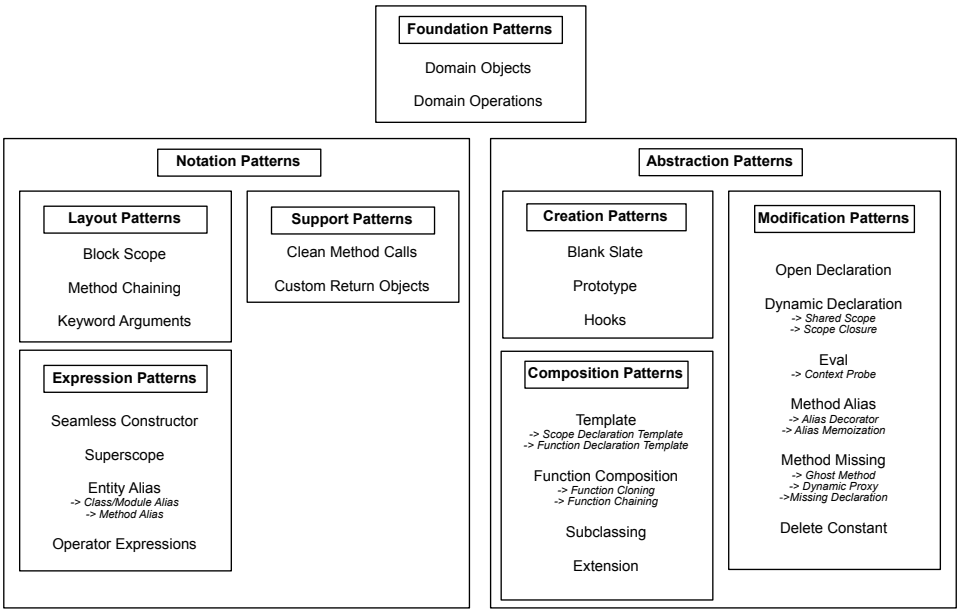

Fig. 6. Pattern Overview

– BLOCK SCOPE: Has your DSL a flat scope where you mangle objects together, or is it difficult to express hierarchies of domain concepts that you create and combine to complex expressions? (►Section 5.1, page 18)

– METHOD CHAINING: Do you use too much command-like expressions that fail to represent the relation of different domain concepts or do not read fluently enough? Excessive (►Section 5.2, page 21)

– KEYWORD ARGUMENTS: Does your DSL suffer from passing to many arguments to functions, confusing the user in which order to pass arguments and what their meaning is? (►Section 5.3, page 24)

– *Expression Patterns*: These patterns target the representation of expressions. Their goal is to simplify and straighten the representation of domain concerns through designing DOMAIN OPERATIONS, and optionally through renaming of built-in or library-added objects and methods.

– SEAMLESS CONSTRUCTOR: How to avoid object instantiation when this is not a domain concern?

– SUPERSCOPE: How to refer to objects that exist in another scope without introducing cumbersome and cluttering variable references?

– ENTITY ALIAS: How to rename built-in classes and external libraries to be consistent with the domain?

  – *Class/Module Alias*: Alias a class or module.
  – *Method Alias*: Provide an alias for a method.

– OPERATOR EXPRESSIONS: How to use operators like +, ∗, & in DSL expressions?

– *Support Patterns*: Further contribute to the efforts of the other patterns with unique effects to simplify expressions.

– CLEAN METHOD CALLS: How to eliminate the clutter of parentheses and other literals for method calls in the DSL?

– CUSTOM RETURN OBJECTS: How to overcome the limited vocabulary for accessing data in multiple structured objects?

4.3.3 *Abstraction Patterns*

Additionally to the foundation and the notation patterns, the abstraction patterns are used too. These patterns help to implement and incorporate the DSL's semantics and abstractions into the host language. They are structured into three groups too.

- *Creation patterns*: These patterns explain how to initially create new objects with properties that are different from common objects.
  - BLANK SLATE: How to provide objects with a minimal set of methods so that arbitrarily named methods can be added?
  - PROTOTYPE: How to create new objects without using the instantiation process to save computing time?
  - HOOKS: How to introduce more-domain specific behavior at pre-defined program execution places like class, method, and attribute declaration?
- *Composition patterns*: Are concerned with the declaration and instantiation of new objects and thereby reuse functionality that is already available in the program.
  - TEMPLATE: How to form mutable string representations of code that facilitate runtime creation, modification, and execution?
    - *Scope Declaration Template*: Declaration of modules and classes.
    - *Method Declaration Template*: Declaration of method definitions.
    - *Method Call Template*: Declaration of method calls.
  - FUNCTION OBJECT: How to represent (defined) application behavior in one uniform, composable form?
  - SUBCLASSING: How to pass method declarations and variables from one class to another, even at runtime?
  - EXTENSION: How to add functionality stemming from different sources into classes, instances, and singletons?
- *Modification patterns*: Are concerned with taking an existing object and modifying it to provide more domain-specific behavior.
  - OPEN DECLARATION: How to change a class or module after its initial declaration by using common declarations?
  - DYNAMIC DECLARATION: How to change a class or module after its initial declaration by using metaobjects or similar techniques?
    - *Shared Scope*: Share the scope of multiple declarations between each other, providing access to local variables for including them in declarations.
    - *Scope Closure*: Conserve local variables in declarations, completely hiding the variables from any outside access.
  - EVAL: How to execute declared code, specified in one scope, at any other scope?
    - *Context Probe*: Execute code in the context of another object to check object properties.
  - METHOD ALIAS: How to transparently change the behavior of an existing method while preserving the old behavior?
    - *Alias Decorator*: Add functionality around an existing method.
    - *Alias Memoization*: Replace a method implementation with a fixed return value to save computational time.
  - METHOD MISSING: How to enable an object to answer arbitrary method calls and to forward the calls to other methods or define called methods on the fly?
    - *Ghost Method*: Depending on the method's name, return a value to the caller that simulates a complete method call.
    - *Dynamic Proxy*: Forward the method call to another module or class.

– *Missing Declaration*: Check the method name and define methods on the fly.

– DELETE CONSTANT: How to completely delete modules and classes together with their methods on demand?

5. NOTATION PATTERNS

A clear, distinguished syntax helps a DSL to convey its meaning effectively. While the foundation patterns provide the essential objects and methods, notation patterns provide the syntax. During our research, we were surprised about the syntactic variations the analyzed host languages offer. In some DSLs, we found a combination of notations that can even lead to a syntax that does not at all resemble the host language. This chapter captures these notations as patterns. We distinguish into three groups. *Layout patterns* provide the overall structure of the DSL. *Expression patterns* provide syntactic alternations for expressions, especially for method calls, in order for a more domain-specific notation. *Support patterns* further help to provide clearer expressions.

As explained to the beginning of this paper, we only explain the three layout patterns BLOCK SCOPE, METHOD CHAINING, and KEYWORD ARGUMENTS. Layout patterns are providing the general layout of the DSLs. In these forms, individual expressions are further designed with variation and support patterns. These patterns are also powerful enough to design a language with a resemblance to another programming paradigm than the host uses, like the change from imperative to declarative expressions.

5.1 Block Scope

Also defined as Nested Closure [33].

Many domains have a natural hierarchy of objects. This hierarchy should be reflected in the DSL expressions too. For example, when building a structured document like HTML, the DSL should reflect the document's structure with syntactic indentation, and at the same time provide a separate lexical scope for its expressions.

Problem

Has your DSL a flat scope where you mangle objects together, or is it difficult to express hierarchies of domain concepts that you create and combine to complex expressions? A flat scope hides the context and natural connections between expressions, making it hard to spot what the expressions actually do.

Design Goals

– *Notation*: Group related expressions together for a vertical code layout and syntactically express the hierarchy of objects as indented blocks of code.
– *Absorption*: Absorb the most relevant execution information as a closure and use it later to infer contextual information.
– *Compression*: Remove fully-qualified method calls inside a block, and let the execution context be determined by the object or method receiving the block.
– *Abstraction*: Execute the expressions contained in a block in another location than they were defined in.

Solution

Define one object that opens the block, and include all expressions that belong in the object's context. By defining a visual block, the hierarchy and scope of expressions becomes clear.

```
machine = Machine.new("Admantium.com")
machine.owner("sebastian.guenther@ovgu.de")
resource_bundle = Array.new(Resource.new("CPU"), Resource.new("RAM"))
machine.set_monitored_resources(resource_bundle)
```

```
Machine "Admantium.com" do
  owner "sebastian.guenther@vub.ac.be"
  monitor "CPU", "RAM"
end
```

A BLOCK SCOPE is defined by the visual arrangement of expressions in the form of an indented block. Semantically, the outer parts of the block provide a context in which the expressions of the inner part are executed. Therefore, the receiver of the inner expressions becomes clear, which allows to implicitly assume information that would otherwise need to be passed as additional arguments to isolated expressions. In ►Figure 7, we show example expressions for each host language.

```
#Ruby
Machine "Admantium.com" do
  owner "sebastian.guenther@vub.ac.be"
end

#Python
with Machine("Admantium.com") as machine:
    machine.owner("sebastian.guenther@vub.ac.be")

#Scala
Machine("Admantium.com") configure {
  owner("sebastian.guenther@vub.ac.be")
}
```

Fig. 7. BLOCK SCOPE: example expressions in Ruby, Python, and Scala.

In Ruby, blocks can be built with the `do...end` notation or curly braces. The code contained inside this block has two distinguishing properties. First, it encloses its surrounding scope and can serve as a closure for specific execution states. Second, it can be executed in another context, using the variables from the surrounding lexical scope.

In Python, context managers can be used for introducing blocks with the notation `with...as`. Context managers are objects that define the special methods `__enter__` and `__exit__`, which are executed upon opening and exiting a block. Opposed to Ruby, fully-qualified method calls still have to be used inside the block.

In Scala, blocks are introduced with curly braces. They are used to group similar expressions, for example method declarations for classes. They can be applied in method calls too, where the return value of a

block is passed as an argument to the function that “opens” the block. In the following example, the result of `owner("sebastian.guenther")` is passed to `configure`.

Example

```
1 machine = Machine "Auxiliary Server" do
2   type :ec2
3   owner "sebastian.guenther@vub.ac.be"
4   os :debian
5   hypervisor do
6     ami "ami-dcf615b5"
7     source 'alestic/debian-5.0-lenny-base-2009...'
8     ...
9 end
```

Fig. 8. BLOCK SCOPE: example for a `machine` expressions.

Using the MDL to configure various virtual machine properties, the BLOCK SCOPE is denoted by the `do...end` notation in Line 1 and Line 8. The properties are included as statements inside the block.

The block is passed to the `Machine` method. Its definition is shown below, starting with the `def` keyword. The method receives two parameters. The first is the machine's hostname, and the second is the block captured in the variable named `block`. Lines 3–6 check whether the machine already exists, and if not creates a new one. Then, Line 7 evaluates the block in the machine's context.

```
1 def Machine(hostname, &block)
2   machine = Machine.first :hostname => hostname
3   if not machine
4     machine = new()
5     machine.hostname = hostname
6   end
7   machine.instance_eval &block
8   return machine
9 end
```

Fig. 9. BLOCK SCOPE: implementation of the `Machine` method.

Liabilities

– Possibility of code injection: If the block object is dynamically constructed with input from the users, malicious users could exploit detailed knowledge of the application to read its data or perform file system operations. However, Ruby has good support for safe levels, as well as tainted and trusted objects [82], which reduces this threat's potential.

Known Uses

– Ruby
  – Using *Sinatra*, BLOCK SCOPE is applied with the declaration of request handlers, for example `get`, to obtain a declarative expression how a web application reacts to queried URLs (the `get` request handler is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Lines 752–758).

– Using *RSpec*, BLOCK SCOPE is applied for the declaration of example groups that begin with the `describe` keyword and group several test cases (the `describe` method is implemented in rspec-1.3.0/lib/spec/dsl/main.rb, Lines 24–29).

– Python

– We could not find this pattern in the analyzed DSLs. In our PyQL DSL [42], we use the context manager construct to introduce a block. The expression `with Query("sqlite") as query` starts a block where `query` is a local variable. Using this object, methods representing SQL-statements are called, but fully-qualified method calls have to be used.

– Scala

– In *Specs*, BLOCK SCOPE is used to visualize the structure of complex test cases. Test cases start with a literate string describing the system, followed by the `should` keyword and a function. Inside the function, examples are specified by using another string description and the `in` keyword (the `should` method is defined in specs-1.6.2.1/src/main/scala/org/specs/specification/SpecificationSystems.scala, Lines 45–46, and the `in` method in specs-1.6.2.1/src/main/scala/org/specs/specification/BaseSpecification.scala, Line 154).

Related Patterns

– KEYWORD ARGUMENTS: Alternatively to the vertical code layout provided with BLOCK SCOPE, code can be arranged horizontally by suitable grouping and naming of arguments.

5.2 Method Chaining

Also defined as Method Chaining [33].

In some domains, expressions involve a high number of concepts and operations. For example in financial transactions, accounts, identification numbers, currency information, stock information and more need to be expressed. This complexity should be reflected in the DSL too.

Problem

Do you use too much command-like expressions that fail to represent the relation of different domain concepts or do not read fluently enough? Excessive command-like expressions make programs difficult to read and understand.

Design Goals

– *Notation*: Express the domain in a more natural language like style by chaining method calls and their return values together, which also gives a horizontal code layout.

– *Compression*: Use chained methods to transform an input value step-by-step, adopting a more functional-programming oriented style instead of imperative expressions.

Solution

Combine repetitive methods in one sentence-like expression.
Improve readability and understandability.

```
1 app = application("Redmine")
2 app.set_server("Admantium")
3 app.deploy
```

↓

```
1 deploy application Redmine on server Admantium
```

Sentence-like expressions can be designed by chaining methods with appropriate names together in one line. The methods are required to either operate on the same directly passed data or implicit on data visible within the scope where the methods are executed. The readability of the method chain can be enhanced by using methods according to these roles:

– *Context provider*: Set the objects of the expression by storing them in one or several variables of the surrounding scope.
– *Referer*: Methods that just refer to concrete objects.
– *Glue*: Methods that only serve to syntactically ease the expression.

Dependent on the host language, a method chain can take different forms. Because Ruby does not require fully-qualified method calls in the form of "receiver.method", the "." can be left out, allowing to chain method calls together that look like language keywords. In Python, chaining method calls both implies to use the dot notation between method calls and to use parentheses, such as in `method1().method2()`. An exception is the usage of symbolic operators, which is shortly presented in the known uses section. In Scala, METHOD CHAINING requires a mix of class and method declarations to work. See the following figure for examples.

```
1 deploy Apache on server "Admantium.com"    #Ruby
2 deploy(Apache).on().server("Admantium")    #Python
3 deploy Apache on server("Admantium.com")   #Scala
```

Fig. 10. METHOD CHAINING: example expressions in Ruby, Python, and Scala.

Example

An expression that uses METHOD CHAINING, and the later explained CLEAN METHOD CALLS pattern, is shown in the following figure. It is a contrived example how to alternatively express the roll out of applications.

```
1 deploy application Redmine on server Admantium
```

Fig. 11. METHOD CHAINING: combining `deploy` and other methods.

The evaluation of this expression happens from right to left. Following figure shows the implementation of the used methods. At first, we define the referers. `Admantium`, defined in Line 2, is the only non-method in this

expression – it is a constant in the surrounding scope that represents the admantium server. In Lines 4–7, we implement the `Redmine` method to return the corresponding object. Next, we implement the context provides in Lines 10–13 and Lines 15–18. They set the passed application and server values to instance variables. The `on` method is a glue method (Lines 21–24). Finally, the `deploy` method in Lines 27–32 commences the deployment of the machine.

In some methods, we implemented explicit checks that prevent to call the chain in an incorrect order. However, as can be seen, this form is very restricted and cumbersome to implement.

```
1  # referers
2  Admantium = Server.find "Admantium"
3
4  def Redmine(on)
5    raise "MethodChainException" unless on == "on"
6    Package.find "Redmine"
7  end
8
9  # context providers
10 def server(server)
11   @server = server
12   "server"
13 end
14
15 def application(app)
16   @application = app
17   "application"
18 end
19
20 # glue methods
21 def on(server)
22   raise "MethodChainException" unless server == "server"
23   "on"
24 end
25
26 # domain operations
27 def deploy(application)
28   raise "MethodChainException" unless application == "application"
29   real_deploy @application.name do
30     enroll "#{@application.name}/#{@application.category}", :on => @server
31   end
32 end
```

Fig. 12. METHOD CHAINING: implementation of all required methods for the `deploy` example.

Liabilities

– Requires to add several “glue” methods to the surrounding scope, possibly polluting the space with empty methods.
– Limited options to check the correctness of the chain, and especially to check the correct order of called arguments.
– Limited applicability to modify existing libraries because of the required extensive modifications.
– If used with an existing library, requires to change method internals.

– The Law of Demeter [58] states that method should have a limited access to other methods and the object in which they are defined. A DSL that uses METHOD CHAINING requires a careful design choice to localize the method declarations in one scope (see DOMAIN METHODS) and should pollute the global namespace that would break the Law of Demeter.

Known Uses

– Ruby
  – In *RSpec*, users chain `should` and `should_not` expression with an expectations statement like `should_not be_nil` or `should have(3).items`, combining the statements in a clean and readable way (both mentioned methods are defined in rspec-1.3.0/lib/spec/expectations/extensions/kernel.rb, Lines 26–28 and Lines 49–51, which extends Ruby's built-in `Kernel` module that defines global available methods).
– Python
  – The *Should DSL* is a unique extension because it uses OPERATOR EXPRESSIONS (explained later) together with a normal method call to add a new syntactic element to Python. Basically, it allows the same kinds of expressions like RSpec, consisting of an object, a `should` keyword, and an expectation. Here is a short example: `1 |should_not| be(2)`. Inside this expression, the "`|`" are actually method calls that receive the left value and the right value, which is then passed to the `should_not` method. This combination allows METHOD CHAINING (the "`|`" operators are defined in should-dsl-1.0/src/should_dsl.py, Lines 25–38).
  – The *LCS DSL* also uses a form of METHOD CHAINING: Methods with boolean return values are used to structure conditional blocks that determine how to react to different status of the monitored applications [11] (we can not give further details because we do not have access to this DSL's implementation).
– Scala
  – *Specs* uses method chaining in an extreme form, because expressions can mirror complete sentences. For example we create a car object with the name "Nissan Primera". To test whether the car name begins with "Niss" and not "Pors", this expression can be used: `nissan.carname must be matching("Niss*") and not be matching("Pors*")`.
  Similar matchers for non-string objects are included in the Specs library, or can be added for user-specific types (the `must` method is defined in specs-1.6.2.1/src/main/scala/org/specs/specification/Expectable.scala, Line 172, and the matchers `be` and `matching` are defined in specs-1.6.2.1/src/main/scala/org/specs/matcher/StringMatchers.scala, Line 36 and 202).

Related Patterns

– KEYWORD ARGUMENTS: Both METHOD CHAINING and KEYWORD ARGUMENTS provide a horizontal code layout. KEYWORD ARGUMENTS are especially suitable when a complex expression can be refactored to nested subphrases. The additional benefit to METHOD CHAINING is that no "glue" methods are needed.
– BLOCK SCOPE: Instead of a horizontal code layout, BLOCK SCOPE provides a vertical layout that is especially useful to show hierarchies of objects. This pattern is recommended when very complex expressions need to be structured, where the block-like structure helps to disseminate the information. Of course, individual lines in BLOCK SCOPE expressions can be designed with METHOD CHAINING.

5.3 Keyword Arguments

Also defined as Named Arguments [69] *and Literal Map* [33].

Methods implement the behavior of a DSL and are vital language expressions. However, when passing multiple parameters to a method, two problems might occur. First, a fixed order of multiple passed arguments is difficult to remember when using the method. Second, just passing arguments hampers readers of DSL expression to relate the arguments with their intended meaning and relationship to the method.

Problem

Does your DSL suffer from passing to many arguments to functions, confusing the user in which order to pass arguments and what their meaning is? If the order of arguments is not clear, hard to track bugs might occur that frustrate the user.

Design Goals

– *Notation*: Avoid complex object interfaces with methods that receive more than two parameters, and avoid the ambiguity of putting long lists of arguments in the right order.
– *Notation*: Use keyword-value pairs as relationship declarations, to state mappings, or to declaratively express the desired system state.
– *Absorption*: From the list of passed arguments, infer implicit information for providing specialized behavior.
– *Compression*: Compress several singular method calls to a set of key-value pairs passed to one method.
– *Generalization*: By using multiple dispatch of arguments, the existing methods can serve as a focus point when adding new behavior.

Solution

Define key-value pairs with the key as the argument's meaning and the value the argument's value. Using key-value parts clarifies the meaning of arguments.

```
deploy_application("Redmine", "Admantium", "default", "restart")
```

↓

```
deploy_application :application_name => "Redmine",
                  :server           => "Admantium",
                  :configuration    => "default",
                  :server_action    => "restart"
```

Multiple arguments can be passed by using hash expressions where the key denotes the argument's meaning and the value is the argument's value. In some languages, the key-value pairs can be expressed along several source code lines, and therefore provide a similar layout than BLOCK SCOPE.

The following figure shows different notations for KEYWORD ARGUMENTS in the analyzed host languages.

```
weather :temperature => 17, :humidity => 25    #Ruby
weather(temperature=17, humidity=25)           #Python
weather("temperature" -> 17, "humidity" -> 25) #Scala
```

Fig. 13. KEYWORD ARGUMENTS: example expressions in Ruby, Python, and Scala.

Ruby hashes use the notation `key => value`, and they can be created as inline arguments to method calls. In Python, methods can be defined to receive `dict` (short for dictionary) objects. Such methods can be called with a hash like notation. In Scala, methods that receive tuples can be used, where the arguments form a sequence which can be parsed conveniently with case matchers.

Example

An expression of the PCML to deploy an application on a machine is used here. The relevant excerpt is the following code.

```
deploy "Redmine" do
  enroll "Redmine/Database", :on => application_server + auxiliary_server do
    #...
  end
end
```

Fig. 14. KEYWORD ARGUMENTS: the `enroll` statement uses hash keywords to configure deployment options.

The enroll method shown in the following code first receives a name parameter and then a hash. This hash is checked for its contained key-value pairs and appropriate values are set for the installation.

```
def enroll(package, hash, &block)
  dependent_package = package_dep.split("/").last
  add_to_install_list @package.dependency(dependent_package)
  target = hash[:on]
  #...
end
```

Fig. 15. KEYWORD ARGUMENTS: implementation of the `enroll` method.

Liabilities

– Complex combinations of parsing arguments and error checking can make the method declaration cumbersome.

Known Uses

– Ruby
  – *Sinatra* uses the `set` method for configuring options, it receives a single key-value pair and sets the configuration accordingly (the `set` method is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Lines 622–636).
  – *ActiveRecord* uses the built-in `autoload` method that receives a module name as a symbol and a filename in which the module is implemented. Modules configured in this way are only imported when they are called in the program (the `autoload` method is defined in Ruby's core class `Module`, which is part of the interpreter implementation).
– Python
  – In *Bottle*, request handlers are defined with normal Python functions and annotation. The annotations determine which queried URL triggers the execution of a method. The annotation can contain additional keywords, for example `@route('/store/product/:id', method='POST')`. This defines a HTTP POST request with the specified URLs to trigger the annotated method (the `route` method is defined in bottle-0.8.4/lib/bottle.py, Lines 398–443).
  – The *LCS DSL* uses a variant of keyword passing. Instead of using hash structures directly, first strings are passed to the functions and then the values of the variables addressed by this string. Here is an example:

`send_command (discrete ("VALVE1", "ON"))` [11] (we cannot give further details because we do not have access to the implementation).

– For Scala, we could not find this pattern in the the analyzed DSLs.

Related Patterns

– BLOCK SCOPE: The horizontal code layout for key-value pairs can also be represented as a vertical stacking by using BLOCK SCOPE. Using a BLOCK SCOPE instead would mean to implement several additional methods to handle the argument passing.

– METHOD CHAINING: For an alternative horizontal code layout, represent the key-value pairs as methods inside a chain. However, likewise to BLOCK SCOPE, additional methods have to be implemented.

6. PATTERN UTILIZATION

In this chapter, we give a general *pattern utilization overview* that document each known uses of the patterns that we found during our research, then we detail the *pattern utilization case studies* where we select one analyzed DSL per host language and show how the patterns are used in it, and finally show the *DSL design principle support* that the patterns can give.

6.1 Pattern Utilization Overview

The ►Tables 6.1, 6.1, and 6.1 list all found occurrences of the patterns for the DSLs of each host language (please note again that we did not check each pattern per DSL, but documented at least one occurrence if we found any). Apart from some patterns not being available in the host languages, we also see that some available patterns are not used.

Table 6.1: Used patterns in Ruby-based DSLs.

Pattern			Rails	ActiveRecord	RSpec	Sinatra
Foundation		Domain Objects		✓		✓
		Domain Operations		✓	✓	
Abstraction	Creation	Blank Slate	✓			
		Prototype				
		Hooks		✓	✓	
	Composition	Template			✓	✓
		Function Object			✓	
		Subclassing	✓			
		Extension			✓	✓
	Modification	Open Declaration			✓	✓
		Dynamic Declaration			✓	
		Eval			✓	✓
		Method Alias	✓		✓	
		Method Missing	✓	✓		
		Delete Constant				
Notation	Form	Block Scope			✓	✓
		Method Chaining			✓	
		Keyword Arguments		✓		✓
	Modification	Seamless Constructor				
		Superscope	✓			
		Entity Alias				
		Operator Expressions		✓		
	Support	Clean Method Calls			✓	✓
		Custom Return Objects		✓		

We make two conclusions. First, some patterns are very specialized in their application area and are therefore not used for other purposes. For example, BLANK SLATE is only being used in one particular DSL, where it provides bare objects on which DOMAIN OPERATIONS are defined. Second, we also think that some patterns, like METHOD ALIAS, are not used in Python and Scala for one simple reason: In Ruby, this patterns is available with the built-in method `alias`, and because is is defined as an abstraction in the language, Ruby developers use this simple yet

Table 6.2: Used patterns in Python-based DSLs.

Pattern			Bottle	ShouldDSL	Cryptlang	LCS DSL
Foundation		Domain Objects	✓			✓
		Domain Operations	✓	✓		
Abstraction	Creation	Blank Slate				
		Prototype				
		Hooks	✓			
	Composition	Template	✓			
		Function Object	✓			
		Subclassing	✓			
		~~Extension~~				
	Modification	~~Open Declaration~~				
		Dynamic Declaration	✓			
		Eval	✓			
		Method Alias				
		Method Missing				
		Delete Constant	✓	✓		
Notation	Form	Block Scope				
		Method Chaining			✓	✓
		Keyword Arguments	✓			✓
	Modification	Seamless Constructor		✓		
		Superscope				✓
		Entity Alias	✓			
		Operator Expressions			✓	
	Support	Clean Method Calls	✓			
		Custom Return Objects				

Table 6.3: Used patterns in Scala-based DSLs.

Pattern			Actors	Spec	Apache Camel	Scala QL
Foundation		Domain Objects		✓		
		Domain Operations	✓			
Abstraction	Creation	Blank Slate				
		Prototype				
		~~Hooks~~				
	Composition	~~Template~~				
		Function Object				
		Subclassing	✓	✓		
		Extension				✓
	Modification	Open Declaration				✓
		Dynamic Declaration			✓	
		~~Eval~~				
		Method Alias		✓		
		~~Method Missing~~				
		~~Delete Constant~~				
Notation	Form	Block Scope		✓		
		Method Chaining		✓		
		Keyword Arguments				
	Modification	Seamless Constructor				
		Superscope			✓	
		Entity Alias		✓		
		Operator Expressions	✓			
	Support	Clean Method Calls		✓	✓	
		Custom Return Objects			✓	

effective modification. But because it is not defined as a simple abstraction in Python and Scala, although it could be implemented as a global function, it is not used. We think that pattern for which we found no know uses are examples where the existing abstractions of the host language can be enriched. Our patterns capture the essence of such an abstraction, and therefore educate developers about the option to implement and use it.

6.2 Pattern Utilization Case Studies

In this part, we take a focused look on one DSL per host language to show how patterns shape the DSL. We consider RSpec (Ruby), Bottle (Python), and Apache Camel DSL (Scala).

6.2.1 *RSpec*

RSpec is a Ruby-based DSL that supports test-driven development. The DSL combines structuring of test suites with `describe` and `it` blocks that receive natural language expressions, and the expression of code from the applications with `should` assertions. The DOMAIN OPERATIONS trio `describe`, `it`, and `should` provides a convenient language for formulating tests. Also, test executions can be screen printed in such a way that the output looks like a checked specification document.

We show an example RSpec test suite in ►Figure 16. Let us suppose we test a `Car` object with valid and invalid configurations of its color and the transmission type. We introduce two test suites in Line 1 and Line 12. Each one contains a test that is introduced with the `it` keyword. Let us consider the first test that starts in Line 3. We first build a car object with a specific model, the color blue and a manual transmission. Once the car object is created, we test if the configuration is valid by using the `valid?` method that is defined with `Car`. The test in Line 8 uses RSpec's capabilities that extend a tested object with the `should` method. This method receives an assertion about a specific property. In this case, `be_valid` refers to the `valid?` methods, which is executed and its boolean return value is the result of the test.

```
01 describe "Configuration of Cars"  do                                    CLEAN METHOD CALLS
02   it "should allow a Nissan Primera to be configured with three colors" do | BLOCK SCOPE
03     car = Nissan.configure do                                             |
04       model :Primera                                                      |
05       color :blue                                                         |
06       transmission :manual                                                |
07     end                                                                   |
08     car.should be_valid                                                   |
09   end                                                                     |
10 end
11
12 context "Configuration of Cars"  do                                      METHOD ALIAS
13   it "should not allow a Nissan Primera to have automatic gear" do
14     car = Nissan.new :model => "Primera"
15     car.color :silver
16     car.transmission :automatic
17     car.should_not be_valid                          METHOD CHAINING, DYNAMIC PROXY
18   end
19 end
```

Fig. 16. RSpec – test suite example for validating car configurations.

Analyzing these expressions in terms of the used pattern, we can see the following:

– CLEAN METHOD CALLS are used all over the place, for example in Line 1.
– Test suites are introduced by the keywords `describe` (Line 1) and `context` (Line 12), where `context` is an ALIAS METHOD.
– Test suites and example groups are formed using BLOCK SCOPE (Lines 2–9).
– Assertions are formulated using METHOD CHAINING (Line 17).
– METHOD MISSING (*Dynamic Proxy*) is used to formulate assertions that refer to methods of the tested objects (Line 17).

RSpec also uses the abstraction patterns HOOKS, TEMPLATE, FUNCTION OBJECT, EXTENSION, OPEN DECLARATION, DYNAMIC DECLARATION, EVAL, and METHOD ALIAS in various parts of its implementation.

6.2.2 *Bottle*

Bottle is a lightweight and minimalistic web framework written in Python that uses declarative DSL expressions. In essence, conventional methods are used as declaration handlers. The methods receive parameterized annotations that determine which URLs are served by the methods. Bottle processes a request by executing the respective annotated method.

```
01 @route('/')
02 def index():
03     query = session.query(Tweet).order_by(Tweet.date.desc())
04     return template('index', pagetitle = 'Latest Tweets',        KEYWORD
05                              tweets = query[:10])                ARGUMENTS
06
07 @get('/all')                                                     DOMAIN OPERATIONS
08 def all():
09     query = session.query(Tweet).order_by(Tweet.date.desc())
10     return template('index', pagetitle = 'Latest Tweets',
11                              tweets = query.all())
12
13 @post('/add_user')                                              METHOD ALIAS
14 def add_user():
15     username = request.POST.get('username', '').strip()         DOMAIN OBJECTS
16     User.construct(username)
17     redirect('/config', code=303)
```

Fig. 17. Bottle – example for the declaration of URL handlers.

In ►Figure 17, we list the declaration of three URL handlers. Each handler uses an annotation to specify for which URL a particular method is called. The first handler in Lines 1–4 is declared with a `route` annotation, its method returns the index page of the application which shows the latest ten tweets. The second handler in Lines 6–9 uses `get` to indicate that it is called only when a HTTP get request is issued. The `get` annotation is just a METHOD ALIAS for `route`. In the annotated method, a query is executed and then a template is rendered to show all tweets. Finally, the last handler in Lines 11–15 is a `post` request handler that accesses the passed `request` object to read the user parameter. This user parameter is again passed to the constructor of the `User` class, and afterwards the request is redirected to the start page.

We find the following patterns in this example:

– Template calls use KEYWORD ARGUMENTS to pass variables that are available in the template (Line 4).
– DOMAIN OPERATIONS that are indicating HTTP methods (Line 6).
– Alternatively to using `route` for the annotation, the METHOD ALIAS `get` or `post` can be used too (Line 11).
– Finally, DOMAIN OBJECTS representing requests or posts are defined inside Bottle (Line 13).

As indicated in the known uses of the patterns before, Bottle also uses the abstraction patterns HOOKS, SUBCLASSING, and EVAL in its implementation.

6.2.3 *Apache Camel DSL*

Apache Camel is a framework to build service-oriented computing applications [68]. The core library is written in Java, but a Scala DSL was implemented on top of it too. The Apache Camel DSL facilitates to express how components interact, how passed data is transformed between components, and even which rules and constraints are considered in order of processing accepted messages.

Suppose we have a service that identifies the keywords of a post in Twitter and returns a list of similar tweets. The service only accepts a number of requests corresponding with the clients pricing model. Clients that connect to the server with a special identification token (bronze, silver, or gold) are forwarded to high-performance servers, while other requests are handled with the normal servers.

```
01 "direct:requests"  ==> {                                    Operator Expressions
02  to ("request:logger")
03  choice {
04     when (bronze_token?(_)) to ( "api:bronze")               Superscope
05     when (silverToken?(_)) to ("api:silver")
06     when (goldToken?(_)) to ("api:gold")
07     otherwise loadblancing roundrobin {                      Method Chaining
08       throttle(20000 per 60 seconds) to ("request:common_one")  Open Declaration
09       throttle(20000 per 60 seconds) to ("request:common_two")
10   }
11 }
```

Fig. 18. Scala Apache Camel – route declaration example.

This is expressed with the Apache Camel DSL source code in ►Figure 18. The first line identifies the listener that receives the request. We use the symbol ==> to show that the request is forwarded to another service. Each request is first passed to a logging handler. Afterwards, we test if the request contains a `request_token` and the type of this (Lines 4–6). If a token is given, the requests are forwarded to a priority server. If no token is given, requests are forwarded to conventional servers that serve 20000 requests per minute (Lines 7–9). Following patterns are used in these expressions:

– Operators Expressions are used to show the flow of the requests (Line 1).
– Superscope is used to refer to concrete URLs and applications (Line 4).
– Method Chaining combines several Domain Operations to concise expressions (Line 7).
– Because of Open Declaration, operations that are implicitly defined for integers can be used to provide natural value declarations (Line 8).

More abstraction patterns that are used by the Apache Camel DSL are Custom Return Objects and Subclassing.

7. PATTERN LANGUAGE

A pattern language provides a vocabulary that explains the application domain, the problems, the design goals, and the solution offered in the form of patterns [2]. It helps to deepen the pattern knowledge by providing additional ways to access a pattern catalog, such as pattern relationship diagrams [34] or further structuring patterns into distinguished purposes or viewpoints [7][61]. This chapter develops the pattern language for our DSL design patterns. At first, we re-visit the *pattern relationships* and express relationships not only among the notation or abstraction patterns, but also between them. In order to better understand how related patterns can be used in development, we also collect a list of *design questions* concerning typical DSL development challenges. Finally, the *pattern integration* section explains how other patterns can be used with our patterns.

A note on formatting: In this chapter, we typeset other patterns in *slanted text*, and keep CAPITALIZED TEXT for our patterns.

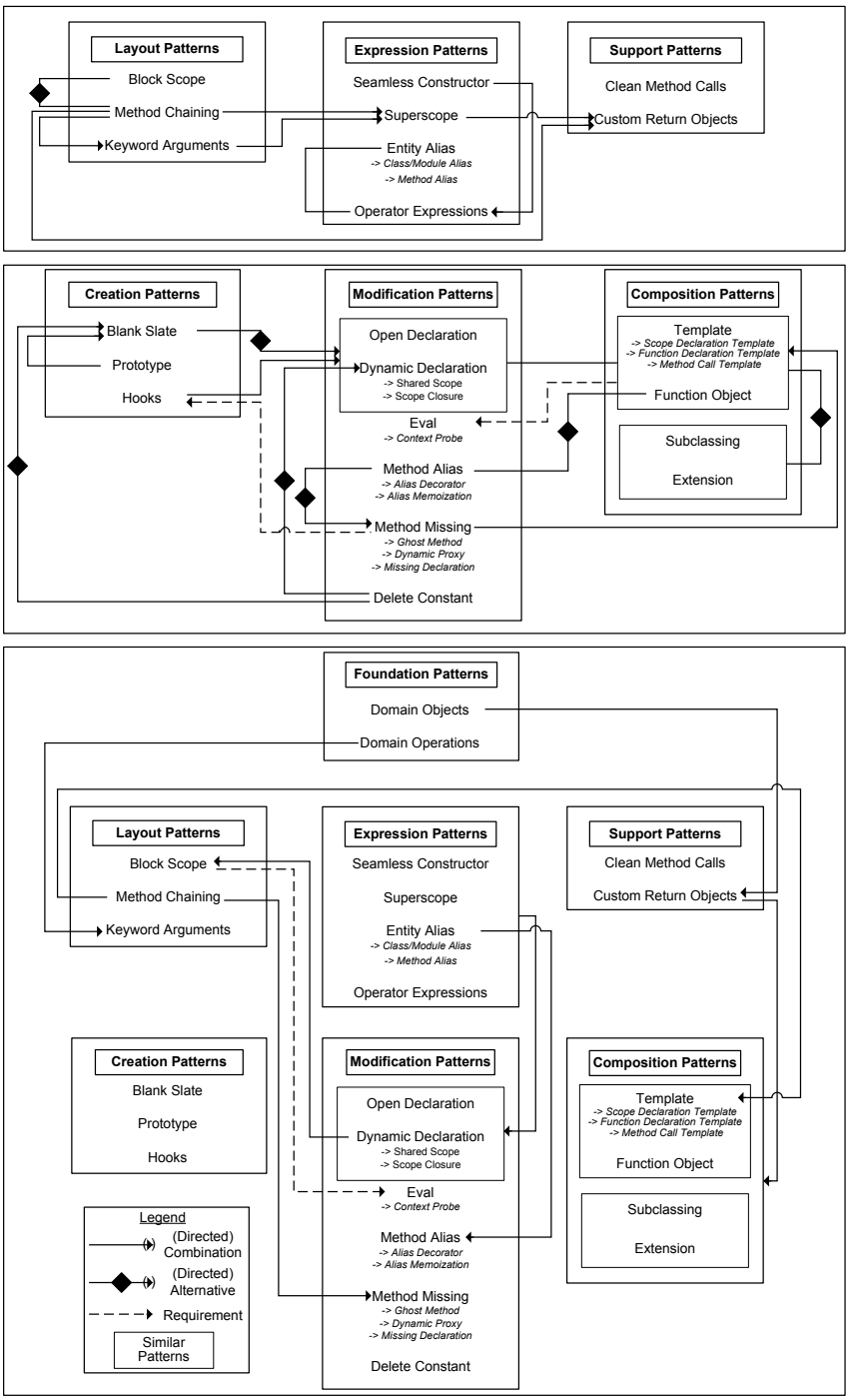

Fig. 19. Relationships among the notation pattern, among the abstraction patterns, and among all patterns.

7.1 Pattern Relationships

The patterns have a rich set of relationships. For reasons of clarity[25], we repeat the relationships diagrams for the abstraction patterns and the notation patterns, as well as adding a diagram that details the relationships between all patterns in ►Figure 19. These diagrams help developers to navigate along the related methods.

The richness of the relationships speak for themselves, but for clarification we provide some further details.

– Data from DOMAIN OBJECTS can be copied to a CUSTOM RETURN OBJECT for restricting the amount of available data.
– DOMAIN OPERATIONS benefit from KEYWORD ARGUMENTS to increase their capability in supporting generalization as an alternative to implement the methods directly.
– METHOD CHAINING can either choose to use TEMPLATES for the different methods, or it can use METHOD MISSING for the method declarations.
– All variation patterns can use OPEN DECLARATION/DYNAMIC DECLARATION to modify built-in classes.
– CUSTOM RETURN OBJECTS can use all composition patterns for defining their structure, and for some cases also the behavior.
– Some DYNAMIC DECLARATIONS may need a BLOCK SCOPE to work.

7.2 Design Questions

The application of the patterns is usually a top-down approach: At first, the DSL's skeleton is defined with the foundation patterns, then the notation patterns are used to design the DSLs syntax, and abstractions patterns help to implement the domain-specific semantics and possible modifications of the host language. This section provide another structure to access the patterns: general design questions that lead the developer through the catalog (inspired by [31][33]). Following questions, roughly ordered according to DSL development steps, provide another guide to the patterns.

– How to define the objects of my DSL?
  ➔ Build DOMAIN OBJECTS and add appropriate setter/getter.
  ➔ Structure the DOMAIN OBJECTS according to the domain hierarchy, and use SUBCLASSING to provide the most common behavior between the objects.
– How to build the methods of my DSL?
  ➔ First, design and implement DOMAIN OPERATIONS in the DOMAIN OBJECTS.
  ➔ During evolution, you may move the DOMAIN OPERATIONS to the global scope for execution, or use a namespace object.
  ➔ If you use BLOCK SCOPE, you can move the execution of the block to the scope where the DOMAIN OPERATIONS are defined.
– What kinds of layout can I use in my DSL?
  ➔ Provide a vertical code layout with BLOCK SCOPE.
  ➔ Provide free-form, line-based expressions with METHOD CHAINING.
  ➔ Provide line-based expressions with KEYWORD ARGUMENTS.
– What is the best way to get a language that does not resemble its host?
  ➔ Combine METHOD CHAINING, SUPERSCOPE, CUSTOM RETURN OBJECTS, and CLEAN METHOD CALLS.
– How to maximize the number of methods I can give my objects?
  ➔ To create objects with a minimal amount of methods, so that you have a maximum space, use BLANK SLATE objects.
  ➔ If existing methods need to have a domain-specific name, use ALIAS METHOD.

[25] Drawing all connections in one diagram, even with different colors, makes it illegible.

– How to add new behavior at runtime?
  ➔ Predefine string TEMPLATES, customize and execute them at runtime (via OPEN DECLARATION or DYNAMIC DECLARATION).
  ➔ Reuse existing functionality of the program with FUNCTION OBJECT (via OPEN DECLARATION or DYNAMIC DECLARATION).
– How to defer the declaration and execution of my DSL expressions?
  ➔ Use SUPER SCOPE to point to concrete entities at the execution of the expressions.
  ➔ Put the expression in a BLOCK SCOPE, and execute the block where the methods are defined (for example DOMAIN OBJECTS or namespace objects).
– In complex expressions, how do I check the right order of expressions?
  ➔ Provide explicit return values that are parameters to the next method. Check the received parameter, and if it is not the expected one, stop the further evaluation.
  ➔ If you use METHOD CHAINING, define local variables as checkers that certain calls were made (ignore the glue methods, confirm the presence of context providers).
– How do I add methods to a built-in type?
  ➔ When available, use OPEN DECLARATION or DYNAMIC DECLARATION to directly modify the type.
  ➔ Use SUBCLASSING to define a custom, built-in type in the global scope, and additionally use ENTITY ALIAS to replace the constant of the built-in type (but better do this in a local scope).
– How to provide better domain-specific notations and abstractions for a library and my DSL?
  ➔ Use ENTITY ALIAS to give existing modules, classes, and methods more domain-specific names.
  ➔ Use METHOD MISSING as a lightweight extension of the library objects. First, implement all additional methods in a custom object. Second, define METHOD MISSING in the original object, but direct method calls to the custom object.
– How to reduce the memory footprint of my DSL?
  ➔ Consider using patterns specifically for optimization, such as BLANK SLATE and PROTOTYPE.
  ➔ If you produce a large number of objects at runtime, then use DELETE CONSTANT to reduce the mount of in-memory objects.
– How to package my DSL?
  ➔ Use a separate namespace to group the DOMAIN OBJECTS and DOMAIN OPERATIONS.
  ➔ Import the namespace into the global namespace.

I need to integrate two DSL – what should I do?
  ➔ Thoroughly study both DSLs, and try to find generalized DOMAIN OBJECTS.
  ➔ Define data that needs to be synchronized between the two DSLs in these DOMAIN OBJECTS.
  ➔ Carefully use OPEN DECLARATION or DYNAMIC DECLARATION to add methods to each DSL that facilitate the DSL integration.
  ➔ Eventually, introduce HOOKS at important points of program execution to automatically aggregate data in the shared DOMAIN OBJECTS.

8. SUMMARY AND OUTLOOK

This paper contributed to the following areas:

(1) *DSL Design Patterns*: We explained three notation patterns, putting emphasize on the concreteness of the patterns. Each pattern presents and in-depth example of Ruby and lists several known uses for 12 analyzed DSLs in Ruby, Python, and Scala. Instead of a mere pattern catalog, we additionally developed a pattern language by providing a deep analysis of the interrelationships between DSL design principles and the patterns as well as a set of design question that guide developers to applicable patterns.

(2) *DSL Design Principles*: The defining properties of DSLs can be expressed as principles that are guidelines for the design and implementation of a DSL. We developed a set of six principles that are aimed at the core task of a DSL, which is the provision of suitable domain-specific abstractions and notations. One set of principles is abstraction, generalization, and optimization, and the other set is notation, compression, and absorption. To define a DSL that embodies these principles, we related patterns and their support for such principles: by choosing patterns appropriately, principle-focused DSLs can be developed.

A particular difficulty that we see with our approach is the required amount of knowledge. Many patterns demand an advanced understanding of the host language capabilities to be realized. While we provided examples and known uses for three host language, which should help developers to better learn from one particular solution, extending the patterns to other host languages simply requires sophisticated background knowledge. Although we think that fully understanding a host language is beneficial in itself and can increase the general productivity of any development, novice programmer may still find it difficult to implement a DSL, especially in host languages that we did not cover.

Another point is the consideration of DSL design principles. The particular principles that we regarded are supportive to each other. It is interesting to think about introducing opposing principles. For example, the verbosity of a DSL is opposed to its compression. In our pattern catalog, these principles could be considered as forces, meaning developers need to balance the peculiarities of the principles. The tensions between different principles could restrict the applicability of patterns, leading to more focused DSLs.

Covering these points means to provide more extensive case studies and evolution experience with a DSL, as well as to consider whether there are some invariants of patterns which are common for a bigger amount of popular programming languages. In the same line of research, we could also gain a better understanding of DSL evolution and the role of patterns, as well as to see how the patterns scale to support DSLs with hundreds of DOMAIN OBJECTS. Another open point is the question of extending the pattern catalog. The nature of patterns makes it hard to say whether a pattern catalog is "complete" – we know of at least two factors that need to be considered. First, the presented patterns are the process of studying existing DSLs and performing own experiments. Conducting more studies and experiments will likely extend this catalog. And second, there are two DSL research directions that are also likely to influence the catalog. One direction is to embed host language independent domain-specific expressions in common host language code, and interact with the host language via compile or runtime transformation to the program's abstract syntax tree (for Java [14], for Smalltalk [70], and for the research language Converge [83]). Another direction are extensible host languages that provide the option to extend the syntax and semantics, with which very sophisticated DSLs could be developed (Kathadin [71] uses a powerful metaobject-protocol for the extension, and Nemerle [74] uses macros). It is interesting to know whether our patterns can be applied to the design of such DSLs, what novel patterns we can find, and how to structure the semantic extension of extensible languages.

REFERENCES

G. Agosta and G. Pelosi. A Domain Specific Language for Cryptography. In *Proceedings of the Forum on specification and Design Languages (FDL)*, pages 159–164. ECSI, 2007.

C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language - Town, Buildings, Construction*. Oxford University Press, Oxford, 1977.

M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-Oriented Programming Languages. In *International Workshop on Context-Oriented Programming*, pages 1–6, New York, 2009. ACM.

B. R. T. Arnold, A. V. Deursen, and M. Res. Algebraic Specification of a Language for describing Financial Products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, Washington, DC, USA, 1995. IEEE.

P. Arpaia, M. Buzio, L. Fiscarelli, V. Inglese, G. La Commara, and L. Walckiers. Measurement-Domain Specific Language for Magnetic Test Specifications at CERN. In *2009 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pages 1716 –1720, Washington, USA, 2009. IEEE.

D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A Domain-Specific Language for Form-Based Services. *IEEE Transactions on Software Engineering*, 25(3):334–346, 1999.

P. Avgeriou and U. Zdun. Architectural Patterns Revisited - A Pattern Language. In A. Longshaw and U. Zdun, editors, *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 431–469, Konstanz, Germany, 2005. Universitätsverlag Konstanz.

J. Aycock. Compiling Little Languages in Python. In *Proceedings of the 7th International Python Conference*, pages 69–77, 1998.

L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6, Vancouver, Canada, 2002. University of British Columbia.

A. Bawden. Quasiquotation in LISP. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Technical report BRICS-NS-99-1, pages 4–12, Aarhus, Denmark, 1999. University of Aarhus.

M. Bennett, R. Borgen, K. Havelund, M. Ingham, and D. Wagner. Development of a Prototype Domain-Specific Language for Monitor and Control Systems. In *Aerospace Conference*, pages 1–18. IEEE Computer Society, 2008.

J. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.

G. A. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Longman, Redwood City, USA, 2nd edition, 1994.

M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383. ACM, 2004.

D. Bruce. What makes a good Domain-Specific Language? APOSTLE, and its Approach to Parallel Discrete Event Simulation. In S. Kamin, editor, *First ACM SIGPLAN Workshop on Domain-Specific Languages (DSL)*, pages 17–35, Illionois, USA, 1997. University of Illinois.

P. Butcher and H. Zedan. Lucinda – An Overview. *ACM SIGPLAN Notices*, 26(8):90–100, 1991.

F. Calefato, F. Lanubile, and M. Scalas. Weaving Eclipse Applications. In A. Gargantini, editor, *Proceedings of the 4th Italian Workshop on Eclipse Technologies (Eclipse-IT)*, pages 29–40, Bergamo, Italy, 2009. University of Bergamo.

B. Cannon and E. Wohlstadter. Controlling Access to Resources within the Python Interpreter. In B. Cannon, J. Hilliker, M. N. Razavi, and R. Werlinge, editors, *Proceedings of the Second EECE 512 Mini-Conference on Computer Security*, pages 1–8, Vancouver, Canada, 2007. University of British Columbia.

T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussels, 2007.

S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, Amsterdam, Netherlands, 2007.

J. O. Coplien. *Multi-paradigm design for C++*. Addison-Wesley, Boston, San Francisco, et al., 1999.

F. F. Correia, H. S. Ferreira, N. Flores, and A. Aguiar. Patterns for Consistent Software Documentation. In *Proceedings of the 16th Conference for Pattern Languages of Programs (PLoP)*, New York, USA, 2009. ACM.

G. Costagliola, A. Delucia, S. Orefice, and G. Polese. A Classification Framework to Support the Design of Visual Languages. *Journal of Visual Languages & Computing*, 13(6):573–600, 2002.

P. Costanza and R. Hirschfeld. Language Constructs for Context-Oriented Programming: An Overview of ContextL. In *Proceedings of the 1st Symposium on Dynamic Languages*, pages 1–10, New York, USA, 2005. ACM.

J. Cuadrado, J. Molina, and M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In A. Rensink and J. Warmer, editors, *Model Driven Architecture âĂŞ Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.

H. C. Cunningham. A Little Language for Surveys: Constructing an Internal DSL in Ruby. In *Proceedings of the 46th Annual Southeast Regional Conference (ACM-SE)*, pages 282–287, New York, 2008. ACM.

K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, San Franciso et al., 2000.

T. Dinkelaker and M. Mezini. Dynamically Linked Domain-Specific Extensions for Advice Languages. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL)*, pages 1–7, New York, USA, 2008. ACM.

S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. Available online http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf, 2004.

R. A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Publishing Company, Menlo Park, Reading, USA, 1996.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, San Francisco USA, 2003.

M. Fowler. *UML Distilled*. Addison-Wesley, Boston, San Francisco, USA, 3rd edition, 2004.

M. Fowler. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, USA, 2010.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Harlow et al., 10th edition, 1997.

S. Gibbs, E. Casais, O. Nierstrasz, X. Pintado, and D. Tsichritzis. Class management for software communities. *Communications of the ACM*, 33(9):103, 1990.

R. Glück and J. Jørgensen. An Automatic Program Generator for Multi-Level Specialization. *LISP and Symbolic Computation*, 10(2):113–158, 1997.

J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing, Indianapolis, USA, 2004.

J. F. Groote, S. F. M. Van Vlijmen, and J. W. C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security (COMPASS)*, pages 57–68. IEEE, 1995.

E. Guerra, J. Souza, and C. Fernandes. A Pattern Language for Metadata-based Frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLOP)*. ACM, 2009.

S. Günther. Engineering Domain-Specific Languages with Ruby. In H.-K. Arndt and H. Krcmar, editors, *3. Workshop des Centers for Very Large Business Applications (CVLBA)*, pages 11–21, Aachen, Germany, 2009. Shaker.

S. Günther. Multi-DSL Applications with Ruby. *IEEE Software*, 27(5):pp. 25–30, 2010.

S. Günther. PyQL: Introducing a SQL-like DSL for Python. In H.-K. Arndt and H. Krcmar, editors, *4. Workshop des Centers for Very Large Business Applications (CVLBA)*, page to appear, Aachen, 2011. Shaker.

S. Günther and T. Cleenewerck. Design Principles for Internal Domain-Specific Languages: A Pattern Catalog illustrated by Ruby. In *17th Conference on Pattern Languages of Programs (PLoP)*, 2010, to appear.

S. Günther and M. Fischer. Metaprogramming in Ruby - A Pattern Catalog. In *17th Conference on Pattern Languages of Programs (PLoP)*, 2010, to appear.

S. Günther, M. Haupt, and M. Splieth. Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Technical report (Internet) FIN-004-2010, Otto-von-Guericke-Universität Magdeburg, 2010.

S. Günther and S. Sunkle. Enabling Feature-Oriented Programming in Ruby. Technical report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, Germany, 2009.

S. Günther and S. Sunkle. Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*, pages 11–18, New York, 2009. ACM.

P. Haller and M. Odersky. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Distributed Computing Techniques.

M. Haupt. Entwicklung einer domänenspezifischen Sprache für das Software Deployment Planning. Bachelor thesis, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, 2010.

K. Havelund, M. Ingham, and D. Wagner. A Case Study in DSL Development – An Experiment with Python and Scala. In *Scala Days*, Lausanne, Switzerland, 2010. École polytechnique fédérale de Lausanne.

J. Heering. Application Software, Domain-Specific Languages, and Language Design Assistants. Technical report sen-r0010, Center for Mathematic and Computer Science, University of Amsterdam, 2000.

P. Hudak. Modular Domain Specific Languages and Tools. In P. Davenbu and J. Poulin, editors, *Proceedings of the Fifth International Conference on Software Reuse (ICSR)*, pages 134–142, Washington, USA, 1998. IEEE.

K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, USA, 1990.

G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, London, United Kingdom, 4th edition, 1995.

D. A. Ladd and J. C. Ramming. Two Application Languages in Software Production. In *Proceedings of the USENIX Very High Level Languages Symposium (VHLLS)*, pages 10–18, Berkeley, USA, 1994. USENIX Association.

P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

F. Latry, J. Mercadal, and C. Consel. Staging Telephony Service Creation: A Language Approach. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 99–110, New York, USA, 2007. ACM.

K. J. Lieberherr and I. Holland. Formulations and benefits of the law of demeter. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

A. Ludwig and D. Heuzeroth. Metaprogramming in the Large. In *Generative Programming and Component Engineering*, volume 2177 of *Lecture Notes in Computer Science*, pages 179–188, Berlin, Heidelberg, Germany, 2001. Springer-Verlag.

M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Survey*, 37(4):316–344, 2005.

G. Meszaros and J. Doble. MetaPatterns: A Pattern Language for Pattern Writing. In *3rd Pattern Languages of Programming Conference (PLOP)*, 1996. Available online http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.8186.

B. Meyer. Principles of Language Design and Evolution. In B. R. Jim Davies and J. Woodcok, editors, *Millenial Perspectives in Computer Science*, pages 229–246, Palgrave, United Kingdom, 1999. Cornerstones of Computing.

D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35:756–779, 2009.

J. Munnelly and S. Clarke. ALPH: A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL)*, New York, USA, 2007. ACM. Available online, http://dx.doi.org/10.1145/1255400.1255404, last access 03/19/2011, 09:47 PM.

L. Nakatani, M. Ardis, R. Olsen, and P. Pontrelli. Jargons for Domain Engineering. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 15–24. ACM, 1999.

J. Neighbors. *Software Construction using Components*. Ph.D. thesis, University of California, Berkeley, Berkeley, USA, 1980.

N. Oliveira, M. Pereira, P. Henriques, and D. Cruz. Domain Specific Languages: A Theoretical Survey. In *INFORUM Simpósio de Informática*, Lisboa, Spain, 2009. Univerity of Lisboa.

M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering (WISEI)*, pages 3–12, Washington, USA, 2003. IEEE Computer Society.

P. Perrotta. *Metaprogramming Ruby*. The Pragmatic Bookshelf, Raleigh, USA, 2010.

L. Renggli, M. Denker, and O. Nierstrasz. Language Boxes. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Proceedings of the 2nd International Conference on Software Language Engineering (SLE)*, volume 5969 of *Lecture Notes in Computer Science*, pages 274–293, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

C. Seaton. A Programming Language where the Syntax and Semantics are Mutable at Runtime. Technical report cstr-07-005, University of Bristol, Bristol, Ireland, 2007.

R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Reading, Harlow, USA, 1999.

T. Sheard and S. P. Jones. Template Meta-Programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

K. Skalski, M. Moskal, and P. Olszta. Meta-Programming in Nemerle. Available online http://nemerle.org/metaprogramming.pdf, 2004.

A. M. Sloane. Experiences with Domain-Specific Language Embedding in Scala. In *2nd International Workshop on Domain-Specific Program Development (DSPD)*, Nashville, USA, 2008.

I. Sommerville. *Software Engineering*. Addison-Wesley, Boston, USA, 9th edition, 2010.

D. Spiewak and T. Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In M. van den Brand, D. Gaševic, and J. Gray, editors, *Proceedings of the 2nd International Conference on Software Language Engineering (SLE)*, volume 5969 of *Lecture Notes in Computer Science*, pages 154–163, Berlin, Heidelberg, Germany, 2009. Springer-Verlag.

D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, Heidelberg, Germany, 2005.

É. Tanter. Contextual Values. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS)*, Washington, USA, 2008. ACM.

S. Thibault, R. Marlet, and C. Consel. A Domain-Specific Language for Video Device Drivers: from Design to Implementation. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, pages 11–26, 1997.

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, USA, 2009.

L. Tratt. Domain Specific Language Implementation via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):pp. 1–18, 2002.

A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):pp. 26–36, 2000.

M. Ward. Language-Oriented Programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

G. M. Weinberg. *The Philosophy of Programming Languages*. John Wiley & Sons, New York, USA, 1971.

D. S. Wile. Supporting the DSL Spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.

U. Zdun and M. Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In A. Kelly and M. Weiss, editors, *Proceedings of the 14th Annual European Conference on Pattern Languages of Programming (EuroPLoP)*, Aachen, Germany, 2009. CEUR, RWTH Aachen.