Nr.: FIN-018-2009

Agile DSL-Engineering with Patterns in Ruby

Sebastian Günther

Very Large Business Applications Lab

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-018-2009

Agile DSL-Engineering with Patterns in Ruby

Sebastian Günther

Very Large Business Applications Lab

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Prof. Dr. Hans-Knud Arndt
Postfach 4120
39016 Magdeburg
E-Mail: hans-knud.arndt@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 04.12.2009

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Agile DSL-Engineering with Patterns in Ruby

Sebastian Günther

Faculty of Computer Science
University of Magdeburg
sebastian.guenther@ovgu.de

Abstract. Domain-Specific Languages are becoming a cornerstone in todays software development processes. Through abstracting twofold - from the narrow details of a programming language, and at the same time from the domain - DSL lead to a coherent representation. This has many benefits: Developers gain a better understanding of the domain, the domain concepts become entities in the program, and solutions to application development problems are done at a higher abstraction level. However, a DSL can never be viewed or used in isolation from other parts of an application. It has to be carefully integrated into the development process and into the usage of other languages. Therefore, DSL need to be engineered with great care.
This report presents a novel approach to DSL engineering which combines lightweight agile development process with implementation patterns. Instead of producing large specifications of syntax and semantic of languages beforehand, the agile process works in small iterations. Using the dynamic Ruby programming language and its entire language infrastructure, the resulting DSL is open to modification and quick dynamic adaption to the changing environment. Inside each DSL, several patterns work to enable this flexibility. The patterns help with Language Modeling (provide executable form of the domain model), Language Integration (how to integrate the DSL into the application framework and with other languages), and Language Purification (means to enrich domain-expressiveness and reduce domain-foreign symbols or tokens). In summary, the developed process and its artifacts, concrete DSL, are tight integrated extensions to current software engineering processes and help with raising abstraction and productivity alike.

1 Introduction and Research Scope

Domain-Specific Languages (DSL) are languages tailored for a specific application area [21]. They use domain-specific notations and abstractions [37] to represent knowledge of a domain in the form of a language. Landin introduced the idea of specific languages [26], and Bentley continued with describing little languages for specific problem domains [3]. Since then, DSL have matured into a common software engineering technique. A good overview of all research topics in the DSL field can be found in [37].

Of particular interest in this paper is the question of how Domain-Specific Languages can be designed and implemented. But there are different types of DSL, which we distinguish.

- **Origin** *External DSL* are developed as separate languages outside of the scope of existing languages. Their syntax and semantics can be chosen freely, but require to de velop and maintain a language infrastructure consisting of interpreter, compilers and linkers. *Internal DSL* instead are based on an existing host language. Their syntax and semantics is predefined and restricted, but can be customized in the boundaries of the host language. On the gaining side, the existing language infrastructure, including tools, can be reused.
- **Appearance**: *Textual DSL* emphasize textual characters and symbols. *Graphical DSL* provide arbitrarily graphical symbols.

Our research focus are textual internal DSL - and whenever we speak of DSL in the following, we mean this specific type. We use the dynamic programming language Ruby as the host language. Ruby is a fully object-oriented dynamic programming language with "duck typing" - Ruby has no type-concept, but sees the method an object responds to as the "type" of an object [36]. It features rich metaprogramming facilities and many options to reduce the syntax of its expressions. Our empirical experiences have shown Rubys great capability to design DSL. Also, various DSL stemming from domains such as database-connections, restful web-services, markup languages, and testing, exist in the form of opensource programs.

This report collects and refines our current experiences with implementing Domain-Specific Languages. Up to now, we have designed a number of DSL, from which two are selected for this paper: One for configuration of product lines [18], and one for feature-oriented programming [19]. We now want to generalize our experiences and design a process how to engineer internal DSL with Ruby. The process we present here has an agile nature. Steps like Domain Design and Language Design are supported with concrete DSL expressions as test cases for implementation. Each iteration implements small parts of functionality. The inner strength of the designed DSL are the patterns and idioms used for their implementation. We discovered three different abstraction level in language engineering to which patterns can be applied: Language Modeling (provide executable form of the domain model), Language Integration (integrate the DSL into the application framework and with other languages), and Language Purification (enriches domain-expressiveness and reduce domain-foreign symbols or tokens).

The herein described process has a number of advantages. At first, it is domain independent - the process does not prescribe any particular domain, but allows to model and design DSL for every domain the creators wish to have a DSL for. Second, it nicely integrates into current software development processes to utilize DSL as a solution to development challenges. Third, since we describe internal DSL, the existing language infrastructure, including compiler, interpreter, IDE, optimizer, libraries and other DSL, can be reused. This allows focusing on language design, not compiler or transformation implementation. And fourth, this approach keeps the DSL "fresh" and dynamic, making it easy to adapt to new environments. In summary, this approach enables internal DSL, which are designed with a scalable and predictable process, to become a corner-

stone in solving today's software development challenges. And although we are using the Ruby programming language, we are certain that the process and the majority of patterns are supported in other dynamic programming languages as well.

In section 2, we will first take a closer look at existing work on language engineering for Domain-Specific Languages. We then continue with an explanation of our DSL engineering process. Section 3 presents the catalog of language engineering patterns. The language engineering process and the usage of patterns is explained with two examples in section 4. In the discussion section 5, we explain how the language engineering patterns fit into existing pattern types, further process augmentations, and how much language independence from Ruby the process possesses. Finally, a summary concludes this report. In this report we apply the following formatings: *keyword*, `language expressions or entities`, *patterns*.

2 Engineering Domain-Specific Languages

Literature about engineering Domain-Specific Languages is divided into the different DSL types. Engineering of external DSL is exemplified in [5], [33], [30]. A recent trend are graphical DSL, e.g. [6] and [23]. These studies target external DSL and thus only have limited use in this report. But they provide a generic process how DSL can be engineered at all. Although their details differ, we can summarize them as follows:

1. Analyze the domain and create a domain model.
2. Define the target language requirements. This includes the concrete syntax, the DSL type (graphical, textual) and overall code-generation criteria (host language, surrounding language framework).
3. Define and implement the necessary code generators. This includes analyzing the host language, the target language, and mapping between their expressions.
4. Generate the application code, check for the correctness of code transformations and use the code in production.

Compared to their external counterparts, internal DSL have a number of advantages. Most important is that internal DSL have what we term a *language infrastructure*. Compilers and interpreters can be reused, and even advanced support tools, like IDEs and code optimizers, are ready to be used with little or no modifications. While internal DSL can use such a language infrastructure, external DSL are usually required to build one for them. Although tools like those suggested in [6] and [23] provide support for writing transformations, developers still need to map input to output transformation manually. In summary, steps 3 and 4 require much less effort when internal DSL are designed.

When we developed our two DSL, one for modeling software product lines [18], and the other one for feature-oriented programming [20], we followed a process only partially influenced by the above steps. Instead, this process used

agile development practices. We started with defining required behavior of our DSL implementation or available language expressions, and implemented these requirements. In small iterations, the behavior was extended. All the while, constant refactoring kept the codebase clean [29] and extensible. The internal quality of the code was high, but in retrospection, we missed the opportunity to provide a well-founded scheme of the DSL and its potential integration with other frameworks and libraries. However, once we saw the need to integrate the DSL, we had no problems in doing so - because the used approach had a thorough foundation.

Another building block which we used unknowingly are *patterns*. In both DSL, we reused certain mechanisms to provide an executable model of the domain, to extend DSL functionality in a modular way, to enable a high expressiveness of the DSL, and to provide different scopes where DSL expressions can be used in programs. Unknowingly, we encountered problems and developed solutions to them - and this is what patterns are about [14].

With this background, we want to develop an agile process for engineering internal Domain-Specific Languages. We combine reflections upon our experiences, exploration of literature about external DSL (as mentioned) and internal DSL [10], usage of existing DSL, and a better understanding of the importance of DSL usage and DSL engineering in application development. The principles guiding this process are (1) *well-founded scheme* of the DSL and the environment it is used in (other frameworks, DSL, technology), (2) *agile steps* providing per iteration just the required functionality, (3) *constant refactoring* to keep the codebase clean and extensible, (4) *pattern knowledge* to have a full understanding of problems and their solution in language engineering, (5) *open form* so that each step takes a specific form dependent on the domain, technology, language and development goals. In the following, we detail the three steps of our process: *Domain Design*, *Language Design*, and *Language Implementation.*

2.1 Domain Design

The first goal is to develop a deep-founded understanding of the domain for which the DSL is to be designed. This step begins with collecting various handbooks, documentation, systems and general stakeholder expressions - this is called *domain material.* The material is studied to produce either formal or informal expressions about the domain. One form is e.g. to use variability and commonality analysis and collect statements in natural-language about the domain [10]. Other forms are so-called domain engineering techniques like FODA (Feature-Oriented Domain Analysis) [11]. If there is no domain material, but only experts having the required knowledge, creative techniques like brainstorming or more formal questionnaires, checklists etc. [9] are usable. We emphasize the importance of this phase. A profound understanding (not necessarily a “complete” specification!) of the domain protects against undesired changes in later steps. Special attention should be given to seemingly contradicting statements - they point at misunderstandings of the domain.

The gained knowledge is then refined to a *domain model*. A domain model consists of the concepts, attributes and their relationships to each other. The collected statements contain singular and compound expressions about the concepts and the relationships. Problematic are possible *language defects* like synonyms, homonyms and more [27]. The domain needs a clear and disambiguate representation so that all stakeholder understand the same concept. It is essential to form coherent knowledge about the domain. While this material defines the *static structure*, we must also model the *dynamic structure* of the domain. This step regards the status of the domain concepts, respectably the *domain objects*, and how they interact with each other with operations. Following the open form principle, the domain model must take a form specific in and useful to the individual engineering process. One suggestion is to use the UML class diagram for the static structure of entities, attributes and relationships, and the state diagram to represent the different status of the domain.

The Domain Design phase performs the whole DSL engineering. The gained knowledge and especially models of the static and dynamic structure are the input to the next phase.

2.2 Language Design

In the Language Design phase, we develop the syntax for a language in which the domain concepts, attributes, relationships and operations can be expressed. Of immediate attention is the language we are designing in. The syntactical constraints of the host language can be a burden to the DSL. Tokens and expressions, which have no meaning in the domain, but are required by the host language (e.g. semicolons, certain brackets, statement modifiers), weaken the *language expressiveness*. It is important to know these limitations beforehand.

Then we begin to formulate expressions in the domain. Expressions need to be valid statements in terms of the host language. Two principal approaches are available. The first one is to design expressions without the host-language in mind, and to make them host-language compatible afterwards. The second approach works vice versa - taking host-language expressions, and simplifying them to increase the language expressiveness. A useful metaphor is that of a *language game*. The philosopher Wittgenstein used language games to determine the grammatical correctness of expressions [25]. Such language games can be used with a compiler or interpreter. If an expression raises only semantic errors, then it is a syntactical valid expression of the host language. This step is repeated until a form of the language has been found. All working example expressions are collected to start the Language Implementation.

2.3 Language Implementation

Having the target expressions available, the task is now to implement them. The basic process of this phase is the *agile process* mentioned in the current section's introduction. It uses a form of *behavior driven development*: First provide a test, and then its implementation. This test can be any language characteristic. In

the very beginning of the implementation, we can easily use *example expressions as test cases*, and build an implementation which has representations of the objects and operations of the expression. Tests are always written first. After passing the test, the existing code is refactored - with the goal of providing a minimal implementation. Successive iterations continue with writing the next test to extend the parsable expressions or other language capabilities. We applied this process in both of our DSL and used *RSpec*[1], a behavior-driven development library, for the tests.

While using this process, we worked on three different abstraction levels. These levels came first to our awareness when we tried to identify and group the patterns which we unknowingly used in the implementation process. As we are aware now, it is advisable to plan the used patterns beforehand since they influence each other. The abstraction levels are the following.

- **Language Modeling** The very beginning of implementing a DSL is to design a implementation of the domain model. This model names concepts, attributes, and operations in the domain. All names form the vocabulary of the language, and they should form the basic structure of the DSL. The task is to map the domain model into a suitable implementation using object-oriented mechanisms.
- **Language Integration** Although a standalone DSL has a value on its own, the option to use with other DSL, libraries and frameworks provides even more benefits. Mechanisms of integration have to be found in order to maximize the effectiveness of the DSL.
- **Language Purification** Syntactical constraints of the used host language are a burden to the DSL. Tokens which have no meaning in the domain, but are required by the host language, weaken the *language expressiveness*. Language Purification is the task of eliminating non-domain relevant tokens by providing syntactical improvements or alternatives - thus raising the language expressiveness.

2.4 Summary

With the DSL engineering process, developers first gather knowledge of the domain, then collect a number of desired language expressions, and finally implement the DSL. The process has an agile nature - phases are encountered in iterations, and each time deepens the knowledge and understanding of the domain. Using a behavior-driven approach guarantees a complete test suite which enables continuous refactoring to keep the code base clean and minimal. Patterns help to structure the DSL according to three abstraction levels. Finally, due to the open form principle, each step embodies the notations, forms and models which the DSL engineers are best accustomed to. This helps to implement and use the DSL successfully. In ►Figure 1 we see a graphical summary of the DSL engineering process.

[1] http://www.rspec.info/

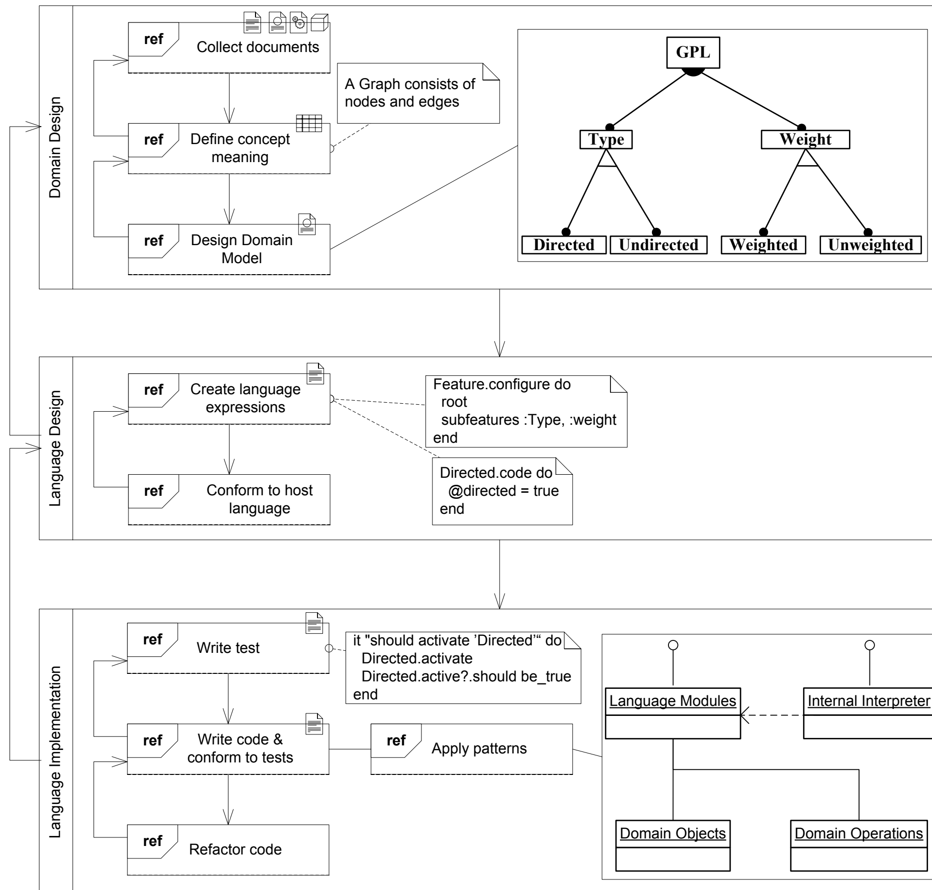

Fig. 1: DSL engineering process

3 Patterns in Language Engineering

The original idea of patterns was introduced in the book by ALEXANDER ET AL. The authors introduced the idea to capture problems and their solutions as building mechanisms for both large and small structures [1]. Combining patterns leads to elaborate architectures. Although ALEXANDER ET AL. spoke of patterns for town and cities, their ideas expanded to other domains as well. One of the first work on patterns in computer science, by GAMMA ET AL., defined that a pattern "... systematically names, explains, and evaluates an important and recurring design in [...] systems" [15]. Each design targets a specific problem in application development - like to provide alternatives for different sorting algorithms or creating customized objects - and a abstract definition of their solution so that they can be reused in different contexts [14]. Thus, patterns are a way to record mature and proven design structures [8]. Patterns have to be distinguished from *idioms*. While patterns are language-independent, idioms are language specific abstractions from problems and can not be used outside their language [11].

Patterns are usually described with a structure that names and classifies each one, describes its intent and motivation, details structure and the participants, and discusses examples [15] [14]. This broad description is chosen deliberately. By discussing various aspects of each pattern, the context in which to apply it successfully becomes known to the developers. But patterns can not prescribe where to use them - their concrete form must always be crafted per context and per application. These thoughts are summarized as "capturing the intent and map it to an application", which is the task of pattern users according to GREENFIELD ET AL. [17].

Still patterns help more then that by establishing a vocabulary which can be used to describe complex architectures [14]. Eventually, this evolves to a *pattern language*. A pattern language details the relationships between different patterns to use them more systematically [17]. Although patterns are usually independent of each other, one choice can lead to another, closely related pattern [14]. Elaborating complex relationships can result in whole maps as shown in [2]. Such views on systems and pattern dependencies provide a rich vocabulary.

In order to use patterns effectively, some requirements must be met: a) the opportunity for the pattern must be understood, b) the elements of the pattern have to be mapped to the application, c) the pattern is restated in the context of the application, and d) the evolving software must keep its links to the patterns [17]. The core requirement is that the developer knows of the patterns and has enough experience to apply them.

In summary, patterns are means to solve recurring problems with common solutions. While we implemented two DSL on our own, we were unwarily using patterns to solve language engineering problems on the three abstraction levels proposed in 2.3. For the abstraction levels Domain Design and Language Design, patterns were used, but the Language Implementation and Support was done with language specific idioms. To avoid to refer to "patterns and idioms" all the

while, we will call them pattern uniformly, and refer specifically to idioms if we need to distinguish them.

For finding patterns and idioms, we did not stick to our empirical gained knowledge alone, but also studied the open-source DSL HAML[2] (HTML), SASS[3] (CSS), DataMapper[4] (database connector), and Sinatra[5] (web application framework). Further analyzed material is [36], [13], [31], and [7].

Patterns are explained with a structure resembling those suggested by FOWLER [14].

- **Name** Name of the pattern
- **Intent** Short summary
- **Form** Presentation of the patterns appearance, usually an example
- **Description and Discussion** Full explanation of the pattern and usage guideline

3.1 Language Modeling Patterns

1. **Command**

Provide objects which represent expressions of the DSL

```
 1 class Command
 2   def execute
 3     raise "MethodNotImplementedError"
 4   end
 5 end
 6
 7 class FileBackup < Command
 8   def initialize(filename)
 9     @filename = filename
10   end
11
12   def execute
13     content = File.read @filename, "r"
14     backup_filename = @filename + Date.today.to_s
15     File.open(backup_filename, "w") { |file| file.write content }
16   end
17 end
```

The command pattern, as originally introduced in [15], defines an abstract `Command` class which has an `execute` method. Objects representing concrete commands should subclass the `Command` class and implement the behavior. Commands are executed by calling the method `execute`.
This pattern allows implementing domain functionality quickly, but the resulting form has several drawbacks. First of all, requiring a call of `execute` does not improve the language expressiveness. Second, combining commands to larger expressions is not possible - each of them does only one thing. And

[2] http://haml-lang.com/
[3] http://sass-lang.com/
[4] http://datamapper.org/
[5] http://sinatrarb.com/

third, since each command requires an own class, extensive amount of commands also bloats the programs namespace with too many objects.
In total, the command pattern should be used by small DSL where quick results are necessary. Furthermore, it may be possible to use some Language Purification idioms to improve the language expressiveness. When one thinks of the agile process to engineer DSL, we also see that any DSL can start as a command pattern, and be improved later to a more robust form.

2. **Domain Objects**

Use classes and modules to provide all domain objects

```
1  module Feature
2     ...
3  end
4
5  class Search
6    is Feature
7  end
8
9  class Algorithm
10   is Feature
11 end
```

Classes and modules can have arbitrary names - with the restriction of a leading capital letter and no whitespace. This makes them perfect for representing domain objects. Attributes can e.g. be defined as instance variables, or yet more comfortable with the `Module#attr_reader` metaprogramming method. The backbone of a consistent and expressive DSL is set with this pattern easily.
To the many benefits of this patterns belongs that the whole domain is expressed with its terms being objects themselves. Every programmer directly works with these named entities - so he gains his domain understanding while programming. In reducing the potential for misunderstanding, the application is more likely to do that what it is intended to do. Another benefit is that Ruby's flexible syntax makes combining objects in expressions straightforward. And even more so, it encourages developers to do that, thereby leading to an explorative form of the mentioned language games.
There is not really a disadvantage to this approach. Of course, we need to provide class definitions. But like shown in the above example, these definitions are very minimal.

3. **Domain Operations**

Implement methods which reflect domain operations

```
1 module Feature
2   def configure(property = String, value = object, *args = [], &block)
3   ...
4   end
5 end
```

Domain objects are not static entities. Their status can be changed, and they can access and change other objects too. These changes take the form of domain operations. A straightforward approach is to implement the operations as methods of Ruby objects. Any interaction between the objects can be modeled in such a way.
Method declaration in Ruby allows many options. Using them greatly enhances the DSL engineers' ability to create the needed "look" of his language. Therefore, we explain the options in great detail. Consider the following method declaration.

```
1 def test_method(default_string = "", default_integer = 0, *optional,
      &block)
2   puts default_string
3   puts default_integer
4   puts optional.join " | "
5   yield block
6 end
```

This example has three types of arguments.

– **Default values** Provide default values via assignment. If no argument is given, the values take the provided default value.
– **Optional arguments** Use one asterisk with any variable name. This argument becomes an array and receives all exceeding arguments passed to the method.
– **Block** Use an ampersand and a variable name. The method receives a block object, which can be called using the yield semantics[6]. Furthermore, the block is available as a Proc object inside the methods' body.

Take a look at the following example of calling `test_method` and the resulting output.

```
1 test_method "task_num", 10, "task A", "task B", :open do puts "Done" end
2 > task_num
3 > 10
4 > task A | task B | open
5 > Done
```

Combining *Domain Objects* with *Domain Operations* is a best-practice approach for domain modeling in a DSL. Having an executable form of the domain knowledge is a mediator between domain and application development which should not be underestimated. The flexible method declarations

[6] Calling `yield` with a given block, executes the code contained in the block. Arguments to yield are passed as arguments to the block [36].

make it even easier to define methods which take zero or an unlimited number of parameters. Using *Keyword Arguments* further clarifies the operation by being verbose about the arguments structure.
One hint to the type of objects on which *Domain Operations* should be defined. The scope of classes is limited to this class, its subclasses and instances. But using modules allows mix-in any functions into any object. This allows providing DSL expressions at the top-level scope, or at a finer granular level. More details hereto are found in the next section, see the pattern *Language Modules*.

3.2 Language Integration Patterns

1. **Hooks**

Use pre- or self-defined hooks to execute arbitrary code at changing program status

```
1 def method_added(name)
2   substitute(name, :instance) if FeatureResolver.violation
3 end
4
5 def singleton_method_added(name)
6   substitute(name, :class) if FeatureResolver.violation
7 end
```

Any application has two specific call stacks. The first one is called *application call stack* and represents the unique composition of objects and modules that the application provides. The second one is the *language call stack* which is represented with the language-internal objects and methods. Using either pre-defined hooks or self-implemented hooks, both call stacks can be modified. DSL can use these hooks to touch deeply into call execution. These hooks are implemented as normal methods - and can thus be modified to execute arbitrary code.
As an example, our FOP implementation *rbFeatures* [20] uses two hooks to intercept method declaration: `method_added` and `singleton_method_added`. Only if the current feature configuration satisfies the feature model, the method is defined with its normal body - if not, the body throws a custom error telling the user which features activation status prevents the calling. This mechanism uses the *language call stack*, and as thus we were able to use rbFeatures both with the Sinatra and the Rails[7] framework without further modification. More hooks for augmenting the language call stack are explained in [19].
It is difficult to determine what kind of hooks a DSL needs to interact with other applications or DSL. In general, the DSL should provide a place where arbitrary Ruby code can be executed. These hooks are the entry point for any specific implementation which the environment might request. And that

[7] http://rubyonrails.org

is the disadvantage: There is no further advice at how to use hooks for specific situations. They need to be experienced and documented in the future.

2. **Language Modules**

Provide parts of a DSL as reusable modules

```
module Operations
  def + feature
    @temp_active = false
    (+ feature)
  end
  ...
end
```

Modules allow defining methods in one place, and to use them anywhere else: Within the module itself, included inside another module, or included inside a class. All inclusions are just pointers to the one module implementing all functions. By changing this one module, all dependent parts change too.
Modules are the most flexible way to share and combine functionality within Ruby. The modules concepts can be used to flexibilize the implementation of a DSL. Like explained, *Domain Operations* define methods which manipulate the domain objects. If put inside *Language Modules*, these methods become even more versatile and reusable. One usage is to design all operations in modules, and to compose them in different variants. The other usage is to use parts of one DSL and parts of another to facilitate language integration. Also, a process-oriented advantage is visible: Independent language modules can be implemented by different language engineers.
Language Modules can be used to provide a DSL within two scopes. In the *top-level scope*, the DSL provides global available methods, so the `main` object of the current Ruby process has to be extended. In the *precise scope*, any module, class or instance object can implement the DSL. Applicability of top-level scope or precise scope has to be determined by the application or framework wishing to use the DSL.
Allowing arbitrary combinations requires some preplanning. To avoid inappropriate uses, precautions like the *Abstract Methods* pattern (prohibit call of module which needs another module to work) or the *Self-Contained Setup* pattern (initialize module-specific instance variables) have to be used. These requirements could limit the *Language Modules* pattern, but more experience is needed to state this precisely.

3. **Internal Interpreter**

Provide a global interpreter object which receives and evaluates DSL expressions

```
1 Interpreter do
2   reconfigure_route "/twitter", :to_match => :TwitterAPI
3 end
```

The classical *Interpreter* pattern as explained in [15] [31] defines the grammar and expression interpretation for any language with an interpreter object. This object receives expressions and applies its interpretation in the scope of the running program.
If a clean separation of different DSL is needed, or a well defined place where DSL statements are executed, then the *Internal Interpreter* should be considered. The interpreter is a global object which receives a block of DSL statements. Statements are executed to change some status of the application, or to interpret and return values. We can say that the interpreter is the representation of the DSL too. The easiest way to implement this representation is to mix-in *Language Modules* into the interpreter.
The interpreter has no disadvantages - its qualities stems from the way how it is implemented as a DSL representation, including the ability to combine DSL expressions.

3.3 Language Purification Idioms

1. **Keyword Arguments**

Named parameters make understanding methods easy

```
1 fill_in "username", "sebastian", "Remember me on this computer"
```

```
1 fill_in "username", :with => "sebastian", :and_option => "Remember me on
      this computer"
```

Methods called with more then one argument risk being misunderstood. In the context of DSL, this hinders understanding what the language does enormously. Using a literal hash as the parameter to the method call explicitly states the meaning and content of arguments. This helps to resolve ambiguity. Furthermore, if the keys form parts of a sentence which reads like natural language, the readability of expressions is greatly improved.
On the downside overhead in parsing the arguments occurs, e.g. error catching. And sometimes the verbose nature of such method calls may not be appropriate for the specific application.

2. **Block Scope**

Provide a clear context for evaluating statements or stack hierarchical information

```
1 t = title "HTML DSL"
2 h = head
3 h.add(t)
4 ht = html
5 ht.add h
```

```
1 html(head(title "HTML DSL"))
```

```
1 html do
2   head do
3     title "Markaby HTML DSL"
4   end
5 end
```

Ruby supports closures and anonymous blocks of code. They can be defined using `do...end` notation. Specifying code in one place, which is to be called in another, is a simple yet very powerful mechanism. Ruby DSL use this mechanism for a number of reasons:

- **Clear execution context** Giving a statement an explicit place at which it is called enhances readability of the expressions
- **Seamless method extensions** Using Ruby's *yield semantics*, a method providing an iteration can apply a given function (as a closure) and apply it immediately
- **Hierarchical information** Express structured data with a layout using blocks

This patterns introduction showed three examples. They are all using HTML entities as *Domain Operations*. All examples show different ways to express the hierarchy of the entities. Example three uses *Block Scope* - we see how much simpler it is to understand the hierarchy expresses in this example.

Here is another example for the clear execution scope usage. Instead of calling `GPL.root`, `GPL.subfatures` in succession, we put them inside a execution scope to enhance the language expressiveness.

```
1 GPL.configure do
2   root
3   subfeatures :Weight, Directed
4   requires :GPL => "all :Weight, :Directed"
5 end
```

The disadvantage of *Block Scope* is the potential of code injection. Whatever is handled to the method is evaluated within a process and certain user rights. Attacker could exploit detailed knowledge of the application to read its data, or perform file system operations. However, Ruby has a good support for safe levels, as well as tainted and trusted objects [36], which reduces this threats' potential.

3. **Method Chaining**

Statements of chained methods to mirror complex grammar structures

```
1 fl = floor.new
2 pound (fl, 1)
```

```
1 pound_of_floor(1)
```

```
1 1.pound.of "floor"
```

Complex object-oriented method calls for retrieving values of objects eliminate language expressiveness. Providing underscored methods requires method declarations for every possible combination. But chaining methods with the minimal syntax of a point together is probably more readable by the domain experts.
Method chains are implemented by taking every method of one object and return the object with `Object#self` to the caller. Immediately, another method is called on this object, which returns itself, and so on.
On the downside, we need to redefine existing methods, or provide empty methods which are just there for syntactical reasons. Methods could be defined automatically via the Support idiom *Method Missing*. But *Method Chaining* is not used often, and usually, *Keyword Arguments* and *Block Scope* provides better alternatives, especially since they do not require changes to existing methods.

4. **Superscope**

Use strings and symbols to transcend execution scope

```
1 configure :application, :with_server => :Thin
```

```
1 def dynamic_feature_method(name)
2   method = <<-EOS
3   def #{name.to_s}(*args)
4     @features[name].execute
5   end
6   EOS
7   return method
8 end
```

From the global namespace, the current execution trace only knows specific entities. Each method call, each variable, and each constant used in expressions needs to be known in this execution scope. Since explicit dependencies are created, modularization is impacted. One option is to use the classical *Proxy* pattern, introduced by Gamma et al. A proxy object determines what object is to be called. This requires a manual mapping of the arguments to a proxy and the called objects. But Ruby provides another way to transcend execution scope.

Ruby programs are treated as strings. As shown in [19], parts of a running Ruby program can be translated to a string, modified, and evaluated back in another execution scope. By using strings and symbols (which are nothing more then immutable strings always pointing to the same place in memory) code can be defined at one place and used in another.
The first example shows how to refer to a object not existing in the current execution scope - with a symbol naming the entity. Ruby has a built-in *Proxy* for global entities: The method `Object.const_get`. Whatever objects the supplied symbol argument represents - it is returned to the caller. Note that the namespace stack for submodules or subclasses has to be resolved manually.
The second example defines a method which returns a string-based method declaration. The returned method name is that of the given argument, and it possesses a body which accesses the instance variable `@features`. Any object can call this method, and use `Kernel#eval` to actually define this method as an e.g. instance method.
In practice, *Superscope* allows to have a clean local scope and decouple the application together with *Proxy*. Developers should however regard whether objects of one scope should be able to address other objects at all. This could hint at a questionable design needing improvement.

5. **Parentheses Cleaning**

Eliminate parentheses around method calls

```
1 fill_in("textfield", Hash.new[:with, "This is a sample text"])
```

```
1 fill_in "textfield", :with => "This is a simple text"
```

Parentheses around method calls reduce language expressiveness. They are a necessity of most programming languages, but usually do not carry semantic information. Ruby makes this very easy. In most cases, all method related parentheses can be just dropped from expressions. The readability is improved immediately. Only some ambiguous cases, like intermixing hashes and blocks, fail.
Although this is a very easy to apply pattern, its impact is fundamental. To the best of our knowledge, no other successful dynamic language allows to skip method parentheses. This was one of the reasons why we started researching Ruby-based DSL in the first point. The importance of changing programming language expressions to a form which does not resemble a programming language can not be stressed enough.

6. **Boolean Language**

Use natural language for logical operations

```
1 if response.success? && password.correct? && !token.rejected?
2   greet "Welcome User!"
3 end
```

```
1 greet "Welcome User!" if response.success? and password.corret? and not
      token.rejected?
```

Boolean operators are common in programming languages. Conditions have to be formed, entries validated, and more. Ruby provides the standard boolean operators *and*, *or*, and *not*. Normally, the symbolic representations are used. But they have natural language counterparts - the one we just used.
Instead of using the symbols, we can switch to their natural language counterparts. In most cases, this is safe. But in some cases, unexpected behavior may occur. The reason: The keywords have a lower precedence then their symbolic counterparts. In general, this is not a problem, since we assume substantial tests for language expressions. So it seems the potential to improve the language expressiveness outweighs the drawbacks.

7. **Operator Redefinition**

Redefine operators to suite the domain

```
1 ship order(:for_user => "sebastian") + order(:as_replacement_for => 345)
      - order(:date => Date.today)
```

Any domain needs to relate its members to each other, compare them, sort them, and select them out of a bigger set. Naturally, symbols for addition, subtraction and so on come to mind. Many Ruby objects have these operations defined. For example, the `Array` class allows to add instances with a `+`, to define the difference with `-`, and to perform a join as simple as `[1,2,3] & [2] #=> [2]`. The secret is: These operations are methods, not language-internal operators.
In Ruby, the basic operators are just normal methods. See ► Table 1 for a complete listing. This means, any domain object can define them. Here we see a method declaration for `+` from this sections introduction example.

```
1 class Order
2   def + (other)
3     other
4   end
5 end
```

Using such symbols together with *Parentheses Cleaning* in expressions is a good way to improve language expressiveness.

Table 1: Redifinable operators in Ruby (from [13])

Operator	Operation
!	Boolean NOT
+ -	Unary plus and minus (defined with -@ or +@)
+ -	Addition (or concatenation), subtraction
**	Exponentiation
* / %	Multiplication, division, modulo
& | ˆ ˜	Bitwise AND, OR, XOR, and complement
<< >>	Bitwise shift-left (or append), bitwise shift-right
< <= >= >	Ordering
== === != =˜ !˜<=>	Equality, pattern matching, comparison

8. **Custom Return Objects**

Return multiple values with the simplest data store - a custom object

```
1 def test
2   return Array.new(true, body)
3 end
4
5 result = test
6 result[1] #=> body
```

```
1 def test
2   Struct.new(:success, :body)).new true, body
3 end
4
5 test.body #=> body
```

Out of the box, Ruby returns exactly one object. If the language designer wishes for multiple return values, they need to be packaged in a collection, e.g. array or hash. The problem is that internal knowledge of the data structures is required on any caller, which is e.g. difficult to refactor. From the DSL perspective, accessing the second value of the returned array looks also debatable.

The solution is to define custom return objects. Does this mean we need to define inner classes in methods just to return them? No, the special `Struct` object comes to the rescue. Calling `Struct.new` with a set of arguments defines a default class object, with the arguments being instance variables and name of setters and accessors at the same time. Used in an assignment, the struct object is bound to the left-hand value. This object has a new method just like for classes, and can be used in the same manner. So, the example above defines a anonymous struct object which has two instance variables, `success` and `body`, and same named accessors and setters.

This pattern has good potential for an alternative *Method Chaining*, which limits required changes and at the same time binds objects and their return values together. One drawback is that named struct objects pollute the symbol tables, but usually a anonymous use, like above, is suitable.

9. **Aliasing**

Change existing methods to have a more domain-specific name

```
1 alias_method :set_option_to_default, :make_defaults!
```

From a historic perspective, many components of Ruby, such as core classes, libraries and frameworks, have been developed in a time where the idea of providing a custom language to access functionality did not had a hold in the Ruby community. "Normal" object-oriented classes and its methods are required to use the component. But what if we want to use a component directly, or even provide a DSL for an existing application?
With the built-in `Module#alias_method` an immediate and simple change can happen. The method receives two symbols as arguments. The first argument is the name of the old method, and the second argument the new method name. Calling the new method redirects to the old one. This provides more domain-related method names, but neither behavior nor external call structure are open to modifications with this approach. Changing method internals is done e.g. via Ruby's open classes concept, as explained detailed in [19].
To rename existing methods, and to change their implementation, helps modifying existing applications to have a more domain oriented form.

10. **Seamless Constructor**

Create new objects for classes without using the new operator

```
1 Add.new(Lit.new(2), Neg.new(4))
```

```
1 Add(Lit 2, Neg 4)
```

The `new` operator expresses the intent to initialize a new instance of any class. A new object is created and bound to a receiving variable or the current execution scope. Some DSL may need new objects, but don't want to call the `new` operator at all. Since Ruby constants are open to redefinition at runtime, we can overwrite the original constant, while keeping the original alive within the body of the redefinition.
The following example shows how to define a seamless constructor.

```
1 def Add(lvalue, rvlaue)
2   Add.new(lvalue, rvalue)
3 end
```

From there on, just calling `Add` with appropriate two arguments creates a new object. This pattern looks nice on the surface, but can have defects internally. Other objects won't be able to query the original object. We could define another constant pointing to the original, but we would need this modified constant in other places too. This complicates application understanding unnecessarily.

3.4 Support Idioms

1. **Prime Activation**

Execute code only when the file is executed by the interpreter

```
if $0 == __FILE__
  load_first :dsl_interpreter
end
```

Ruby libraries may contain files which need some sort of setup to work properly. If one uses them directly, certain setting may yet have not been defined. *Prime Activation* solves this challenge by adding a block of code to the file which is only executed of the file is directly executed.
In the context of DSL engineering, we may use this if our Ruby program is composed of different DSL. Enabling to use them separately requires loading the appropriate DSL definition first. The example above e.g. loads a *Internal Interpreter* first. However, using the pattern is to question as better and more object-oriented mechanisms for enabling reusability exist, like the following *Self Contained Setup*.

2. **Self-Contained Setup**

Included Modules define required variables automatically

```
module Message
  def self.setup
    "@message = \"Started\""
  end

  module ClassMethods
    puts "Executed"
    def message
      @message
    end
  end

  def self.included(base)
    base.extend ClassMethods
    base.send :eval, Test.setup
  end
end

class Dispatacher
  include Message
end
```

Rubys' modules are the primary way of sharing functionality. One challenge is to design modules which extend objects functionality, but rely on local variables. Any potential object needs to define these variables upfront. This is too much coupling between independent entities.
The proposed solution uses the `Module#included` hook. This method is called on the module, and receives the object it is included into as an argument. The module itself has a setup method which returns any code to be executed on the object. This code could be requiring other modules or classes, define

variables, or methods, and much more. In the example, we use a string and `Kernel#eval` to define the local `@message` variable. In total, this pattern allows to seamlessly compose different modules and their functions into one object without coupling the two too much.

3. Abstract Method

Protect methods against inappropriate usage by raising a default error

```
1 def abstract_method
2   raise AbstractMethodError, "Please Provide an implementation for
       'abstract_method'"
3 end
```

Abstract methods are a mechanism to define an interface to an object or module. When a method is called on an object providing this interface, the method needs to be overridden, or else raises an error. Languages such as Java have a direct statement modified for defining those methods, but Ruby has not. But the mechanism is easy to implement.
The solution is to provide a default method body for those methods which are to be abstract. This body simple raises an error when called. Any object gaining this method, e.g. subclassing or as an included module, has to redefine the method. While this mechanism works, it does not reflect what it does using appropriate keywords. But we could provide a method which is used like this:

```
1 class LoggerAPI
2   abstract :log, :event
3 end
```

4. Method Missing

Define missing methods on the fly

```
 1 module DSL
 2   def method_missing(sym , *args , &block)
 3   case sym
 4     when :app
 5       (class << self; self; end).class_eval do
 6         define_method sym do
 7           block.call
 8         end
 9       end
10     else
11       super( sym , * args , &block)
12     end
13   end
14 end
```

Method Missing is a very prominent pattern in the Ruby community. It is used e.g. in *ActiveRecord*[8], Rails database DSL. Calling a method like `find_by_name_and_familiyname` triggers the *dynamic finder* mechanism which

[8] http://ar.rubyonrails.org/

looks whether the queried attributes exists and if yes, return the value with an internal method call.
Another use case is to define the method that was missing on the fly. Consider the case of calling a method with a supplied block. When the method is not available, it should be defined and when called, it should execute the supplied block. The above example enables this. The method `method_missing` receives a symbol specifying the called method, an array of its arguments, and the supplied block. Inside, we compare the symbol, and if it is `:app`, then we define the method with a body which calls the supplied block.
However, with great power comes great responsibility. `method_missing` should be used with caution and only in the namespace of a framework or a DSL. If multiple frameworks would overwrite the `method_missing` method in `Kernel`, this pattern quickly becomes the antipattern *Monkey Patching* - incompatible redefinitions lead to erroneous code.

3.5 Summary

Engineering Domain-Specific Languages requires working on three abstraction levels. *Language Modeling* is the task to provide a executable form of the domain as objects which implement operation. *Language Integration* considers how a DSL can be integrated with other frameworks or other languages. And finally, in order to have a high language expressiveness and thus foster the DSL success, *Language Purification* helps to eliminate domain-foreign tokens. For each abstraction level, several patterns were presented. Each pattern was described with an example, its usage, and potential disadvantages. Some patterns already named alternatives or corresponding patterns, but the majority requires more usage and development experiences. The current catalog will evolve and grow with future research. We continue with two examples for DSL engineering using the suggested process and patterns.

4 Examples

In the past, we developed two DSL. The first DSL targets the domain of Software Product Lines. It configures an abstract feature model by describing the relationships and constraints features have to each other. The second DSL targets the domain of feature-oriented programming. By making features entities of the host language, we bring a whole software development paradigm to Ruby. Originally, both DSL were developed with a loosely related sequence of slightly modified steps. But in retrospection we found no difficulties in aligning past steps with the current language engineering process' form. Thus, we want to use existent examples as a validation for the process, and accompany the process' evolution with further case studies of DSL engineering.

In the following, both DSL are explained with enough background material to understand their domains and expressions, and then we focus on the process and patterns.

4.1 Software Product Line Configuration Language

4.1.1 Explaining the DSL

The Software Product Line Configuration Language (SPLCL) was our first attempt at creating a Domain-Specific Language. Software Product Lines address the important challenge to structure valuable production assets in a meaningful way to support productivity and reusability [11]. Withey implies the important strategic value of such assets, and he further defines product lines as a "group of sharing a common, managed set of features" [38]. We explain the language following the Graph Product Line example [28], which is depicted as a feature tree in ►Figure 2.

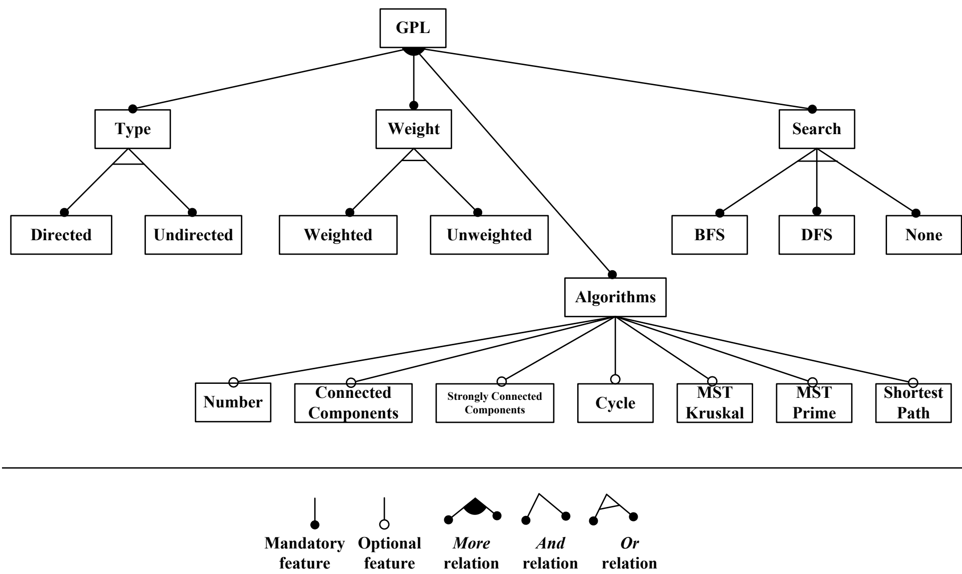

Fig. 2: Feature tree of the Graph Product Line

SPLCL provides modeling capabilities for a complete feature tree. A feature tree is considered to have features of type `root`, `node` and `leaf`. Each feature defines subfeature relationships. Constraints use the keywords `all`, `any`, `one`, `more` and `is` to relate their selection in the tree with the existence or choice from other features.

In the following, we see the definition of the root feature, and explain the meaning of individual statements.

```
gpl_feature = Feature.configure do
  name :GPL
  root
  subfeatures :Type, :Weight, :Search, :Algorithms
  requires :GPL => "all :Type, :Weight, :Search, :Algorithms"
end
```

- Line 1 defines an object `gpl_feature`
- On line 1, `Feature.configure` is the class constructor for features
- The keywords `begin` and `end` on line 1 and 5 define an anonymous block of code that is executed in the context of the receiver
- The feature is named `GPL` on line 2
- This feature is given the `root`type on line 3
- Its `subfeatures` are `Type`, `Weight`, `Search`, and `Algorithms` (line 4)
- `requires` defines that if the `GPL` feature is included, then `all` the features `Type`, `Weight`, `Search`, and `Algorithms` must be included as well (line 5)

Once all features are created, the next step is to configure the `ProductLine`.

```
spl = ProductLine.configure do
  description "The complete GraphPL"
  add_feature gpl_feature
end

spl.add_feature type_feature
spl.add_feature weight_feature
spl.add_feature search_feature
spl.add_feature algorithms_feature
spl.add_feature directed_feature
spl.add_feature undirected_feature
```

It receives a `description` and adds features with the `add_feature` method. The product line has built-in logic for checking if only one root exists, if all named features in the `subfeatures` relationships are included, if all features are connected to each other and if their type corresponds to their position.

The final step is to create a `ProductVariant`.

```
pv_cycle_numbers = ProductVariant.configure(spl) do
  description "Basic variant with algorithms Cycle and Numbers only"
end

pv_cycle_numbers.activate_feature :GPL
pv_cycle_numbers.activate_feature :Type
pv_cycle_numbers.activate_feature :Weight
pv_cycle_numbers.activate_feature :Search
pv_cycle_numbers.activate_feature :Algorithms
pv_cycle_numbers.activate_feature :Cycle
pv_cycle_numbers.activate_feature :Numbers
```

This object receives a `ProductLine` object with its constructor. Only if the `ProductLine` is valid, a new `ProductVariant` object is created, otherwise an error occurs. Furthermore, a `description` can be added. The variant is configured with `activate_feature` and `deactivate_feature`. The current configuration can be checked with the `valid?` method.

4.1.2 Language Engineering

Considering the process, we made the following steps.

- **Domain Design** Three entities are important for the SPLCL: The single *feature*, the *product line* as a hierarchical structure of features, and the *product variant* as a selection of any feature from a product line. Features are modeled in a tree structure, so that both *feature hierarchies* and *feature constraints* can be expressed. Constraints are in the form "if A, then B" or "if A, than all B, C and D". Following, the product line is simply the set of all features, forming a tree representation. The product line is valid if all its features are reachable from each other and if all features are complete. Finally, the product variant is an instance of the product line. It activates certain features and provides a configuration of the product line. The constraints for all activated features must yield true, or else the whole variant is invalid.
- **Language Design** We need a representation of all entities. The `feature` receives a `name`, `description`, a position of either `root`, `node`, or `leaf`, a set of `subfeatures`, and any number of `constraints`. The `ProductLine` would also get a `description`, and could use `add_feature` or `remove_feature`. Also, the product line needs a method to check whether it is `valid` or not. Finally, `ProductVariant` needs to receive an existing product line and a `description`. Users can `activate` and `deactivate` individual features.
- **Language Implementation** Since this language was the very first DSL we implemented, the first iterations merely tested what kind of syntax Ruby offers for making configuration as easy as possible. The most important design goal was to have a minimal syntax - it should contain only symbols of the domain, but nothing which resembles a programming language. After some time, we used *Block Scope* and *Parentheses Cleaning*. This satisfied our syntactical goals. A challenge was to reference features inside a product line without using the constant with which the feature was created first. Here, *Superscope* helped us in decoupling the application. Once the basics were clear, we used extensive RSpec tests for testing each possible variant of the graph product line, e.g. as this one:

```
1  it ' * define the basic variant to include a product line description' do
2    pv_cn = ProductVariant.configure(spl) do
3      description "Basic variant with algorithms Cycle and Numbers only"
4    end
5    pv_cn.activate :GPL, :Type, :Weight, :Search, :Algorithms
6
7    pv_cn.get_all_features.should have(5).item
8    pv_cn.get_all_features.should include :Search
9    pv_cn.get_all_features.should include :Algorithms
10 end
```

In terms of the pattern language, SPLCL has the following form:

- **Language Modelling** *Domain Object*, *Domain Operations*
- **Language Purification** *Block Scope*, *Parentheses Cleaning*, *Superscope*
- **Language Integration** *Hooks*

4.2 rbFeatures

4.2.1 Explaining the DSL

One important challenge of software development is to separate its many dimensions, like requirements, functionality, and technologies, in a way so that the resulting application is still maintainable and extensible. One answer to this challenge is feature-oriented Programming (FOP) [32]. Features, seen as advances in functionality [22], modularize the software along one additional dimension. Instead of being limited to the languages modules or classes, features provide another layer to modularize functionality.

rbFeatures is our contribution to FOP. It is a DSL which brings features as "types" to Ruby. rbFeatures has been extensively covered in [20], and its metaprogramming foundation can be studied in [19]. However, for being concise, we introduce the most important facts about rbFeatures here.

The first step to use rbFeatures is to define features of the program. Any class object becomes a feature by simply including the feature module. With this, the class gets methods to activate and to deactivate itself, to query its activation status, and many private methods for the internal logic of rbFeatures. Once all features have been defined, the next step is to form so-called *feature containments*. Inside a program, particular pieces of source code - whole methods or single lines - belong to certain features. These parts are put inside containments: A normal Ruby block together with a *containment condition*. The containment condition contains a test for at least one features' activation status. Only if the condition yields true, e.g. when the features are activated, then the code gets executed. Conditions like e.g. "if feature A and feature B are active, but not feature C" map to `A + B - C`.

After the second step, the program is said to have been *feature refactored*. The program is now ready for forming any variant as an expression of a particular set of activated and deactivated features. Variants are built either at the initialization of the application, or dynamically at runtime. Activating or deactivating features re-evaluates parts of the source code, thus leading to changed methods, added fields etc. In total, the amount of changes to a working application is minimal: defining features and expressing parts of the program in feature containments. That is all whats needed to enable Feature-Oriented Programming in Ruby. Lets take a look at an example.

```
class Print
  is Feature
end

class Eval
  is Feature
end

class Lit
  def initialize(val)
    @value = val
  end

  Print.code do
    def print
      puts @value
    end
  end

  Eval.code do
    def eval
      @value
    end
  end

  def print_eval
    Print.code { "print #{@value.to_s}|" } + Eval.code { "eval #{@value}" }
  end
end
```

Here we see that two features are implemented: `Print` and `Eval`. In the class `Lit`, we see three methods. The first two, `print` and `eval`, are included in feature containments. The methods are only defined if the corresponding containment condition - in this case a simple test if either Feature `Print` or `Eval` is activated - is true. Another method, `print_eval`, is implemented by default, but returns different values according to the features activation status.

4.2.2 Language Engineering

rbFeatures was developed following these steps.

- **Domain Design** Other FOP solutions express features not as first-class entities of the language, but with various external mechanisms [34] [35]. Since a gap between representation and implementation exists, we wanted represent features as entities of the language. So, a feature would be an object inside our language, which could be activated and deactivated, and express what code in the program belongs to this feature.
- **Language Design** The top entity is the `Feature` itself. It uses methods to `activate` and `deactivate` it. Code belonging to a feature is expressed as a code block, which is handled to the `code` method. The second entity is the `FeatureResolver`. Its only external visible methods are `init` which receives the code of the application, and `reset!` to set all features to the deactivated status again.
- **Language Implementation** We used an extensive RSpec test suite for many dimensions of rbFeatures. Starting point were containments which

included method declaration to test whether they defined methods correctly. We than became aware of checking that methods were defined only in the scope they are supposed to be defined, and not to let a instance method become a class method. Next we added a test suite to ensure the preservation of method type and visibility, and so on. The most recent tests are for the containment condition. Consider the following test which checks whether the expression `A | (B & C) - D` behaves correctly.

```
1 it " * Complex mix with And [Add | (Sub & Div) - Tim]" do
2   Add.activate
3   Sub.activate
4   Div.activate
5   Tim.deactivate
6   ((Add | (Sub & Div)) - Tim).code{true}.should be_true
7 end
```

During implementation to conform to the tests, we applied several patterns. Blocks which surround those parts of the program which belong to a feature is the dominant syntactical form (*Block Scope*). Every class should become a feature. To express the domain, we aliased the `include` method to a `is`, so that we can express what classes are features with the `is Feature` expression (*Aliasing*). We began with a complete module, but later factored out different modules with rbFeatures specific code (*Language Modules*). In order to use natural symbols like `+, -, |`, and `&`, we created a separate module (*Operator Redefinition*).

In terms of our pattern language, rbFeatures is expressed as follows.

- **Language Modeling** *Domain Objects*, *Domain Operations*
- **Language Purification** *Block Scope*, *Aliasing*, *Operator Redefinition*, *Parentheses Cleaning*
- **Language Integration** *Hooks*, *Language Modules*

5 Discussion

In order to complete the understanding and utilization of the process, we discuss how to integrate DSL engineering with the general software development process. The next point is to form a better understanding where language engineering patterns relate to other forms of patterns. We continue with suggesting a enhancement of the current process, and discuss as the last point how the research results of this report can be extended to be used with other dynamic programming languages.

5.1 Integrating DSL Engineering and Software Development Processes

In our experience, DSL allow another kind of solving software development challenges. By raising the abstraction layer and letting developers work with a language that expresses the domain, software reuse is boosted [30] and with it

overall productivity. In our view, a DSL is a tool in solving complex engineering problems. From a viewpoint of solving problems with the right tool, DSL engineering may also be considered as a paradigm. Coplien gives many more ideas how to use multi-paradigm design for application development effectively [8].

But in order to use DSL effectively, some considerations have to be made. The first consideration is how the development process for DSL and for software is related. The answer is straightforward: Both processes follow the universal steps of analysis, design and implementation [16]. In the course of analyzing the requirements of a particular program, it may be discovered that a number of domains have to be considered in the application. A DSL helps in representing the domain inside the program. If the domain is likely to occur in other programs, or already has occurred, then the design of a DSL can be taken into consideration. The software development would go on normally - with the exception that one of its working packages is the design and implementation of the DSL.

The second consideration is how to deploy the DSL engineering process concretely. To the beginning of our process, we stated the open form principle: each process step takes a specific form dependent on the domain, technology, language and development goals. It is open to the DSL engineers what concrete methods and mechanisms to use for representing the domain model, record language expressions, design the language and its status, and documentation. In our view, the methods for designing the overall application should be reused for the DSL - after all, the DSL is software too. Reusing known and working methods improve developers acceptance to engineer DSL and the success rate of doing it.

Additionally to the form, the process' nature is also important. We presented an agile process. The fundamental distinction to other processes, like the waterfall model, is that instead of having large upfront analysis phases, each iteration only analyzes the most important parts of the domain, and implements them. After each iteration, working software exists, which can be modified to meet the succinctly refined requirements of its stakeholder. But what if a DSL should be designed in a software process with a waterfall model? The process remains usable with this setup too. The only restriction is that developers have the whole knowledge about the domain and the language engineering patterns in advance.

The last consideration is how to use DSL in application engineering. In our view, using a DSL in isolation only raises the abstraction level for one domain of the application, but the rest of the application stays at its current level. While this has advantages, we have a very specific view how to maximize DSL usage. In [18], we suggested designing applications using multiple DSL and one base language only. We term this *Multi-DSL Applications*. We argue that when we abstract all domains, technologies, and foreign languages with carefully crafted DSL, we can use the same concept in all different layers of the application. This simplifies specifying, implementing, testing and maintaining the application tremendously. We already worked on this kind of applications, and will provide further explanations in the future.

5.2 Structural Relationships of Patterns

5.2.1 Types of Patterns

General literature on patterns distinguishes abstraction level or responsibilities of patterns in software. This section will explain some viewpoints, and then relate the language engineering patterns to them.

The first distinction is the one proposed by BUSCHMANN ET AL. [4]. He termed three abstraction levels for patterns: *Architecture Patterns*, *Design Patterns* and *Idioms*. In the original meaning, *Architecture Patterns* structure the application into subsystems with well-defined responsibilities. At the second level, *Design Patterns* detail the subsystems inner-structure with components, and defines the interaction of subsystems and the components. *Idioms* are language-specific solutions to build the components.

Next, we discuss the patterns *Creational*, *Structural* and *Behavioral* as suggested by GAMMA ET AL. [15]. These patterns distinguish the kind of problems which occurs in object-oriented programs. Creational patterns suggest strategies to create objects in a decoupled way. Structural patterns elaborate mechanisms to separate responsibilities between objects, and behavioral patterns how the objects collaborate with each other.

The third distinction we want to make is the separation of layers inside an application. FOWLER sees three layers for his patterns [14]. *Domain* patterns express the logic of the domain, like validation of its entities, and how these entities are structured to each other. *Data Source* patterns describe how to link domain entities with databases so that entities are saved and retrieved according to the conditions defined by the domain logic. Finally, interaction is driven by the *Presentation* patterns - the patterns herein explain how to route requests down to the data source and ensure everything is happening according to the domain logic. FOWLER provides additional patterns for *Object-Relational Metadata Mapping*, *Offline Concurrency* and *Session State* to solve layer-specific problems.

When we reflect upon the patterns and the abstraction level they are targeting, we see that BUSCHMANN ET AL. provided vertical patterns, distinguishing abstraction viewpoints on creating whole applications. None of the patterns are idioms, but still they target different abstraction levels. We see Creational and Behavioral patterns, suggested by BUSCHMANN ET AL., on the Design level only, while the Structural patterns can be used in both the Application and Design level. Patterns suggested by FOWLER are on the Design level only. These relationships are shown in ►Figure 3.

5.2.2 Patterns and Language Engineering Patterns

How are the explained pattern and language engineering patterns fitting together? To answer this question, we must see what general place a DSL takes in the application architecture. We consider the design of a web application. The basic layout is the *Model-View-Controller* pattern. This patterns' workflow is that the controller receives a request, interacts with the model to retrieve some

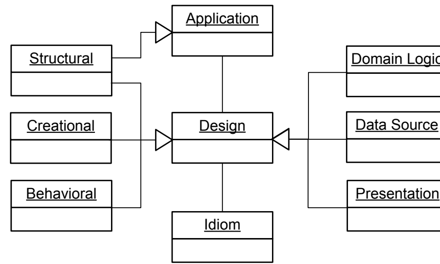

Fig. 3: Pattern relationships

data, and then renders a view. *Model-View-Controller* has three components. For its *model* and its *view* we want to use a DSL, which is depicted in ►Figure 4.

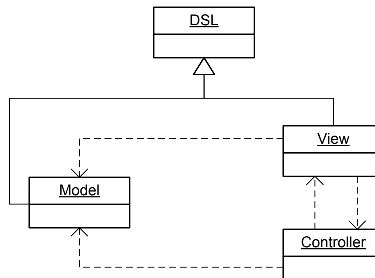

Fig. 4: Model-View-Controller pattern augmented by a DSL

We further elaborate the application, and after some iterations of language engineering for the model, we have a preference for using both *Language Modules* and *Interpreter* for the Language Integration, and several Language Purification idioms. A graphical representation of the pattern language used to describe the pattern is shown in ►Figure 5.

Once we have this representation, we want to express that the component is realized by the DSL. Therefore, we replace the DSL component with the elaborated pattern language, and use *Language Modules* and *Interpreter* as the interface to the DSL which is used by the model component. This is shown in ►Figure 6.

Generalizing from this example, we see that a DSL can abstract any component of an application. Greenfield et al. expresses that a pattern language is an incubator for a DSL [17]. Once the patterns abstraction becomes accepted

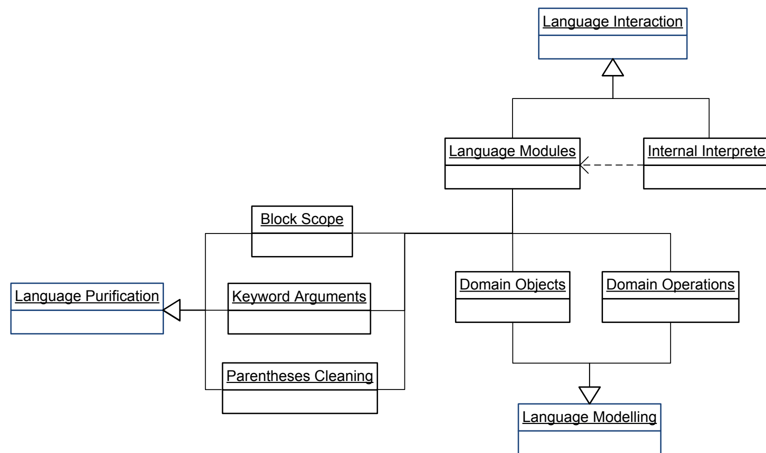

Fig. 5: Pattern structure for a DSL

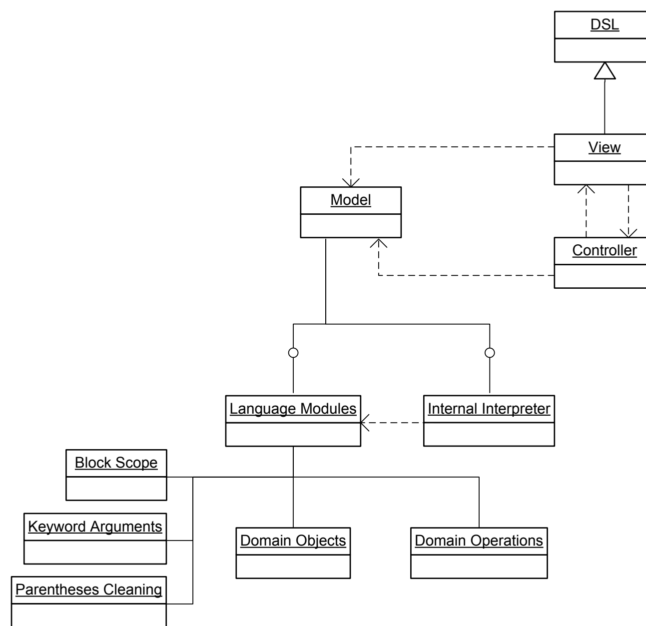

Fig. 6: Realizing the model component with a DSL

by the community, they will be realized within the language. In our case, this realization has the form of a DSL. But can a DSL also implement a whole pattern?

Lets analyze the *Sinatra* web framework. On top of *Rack*[9], an interface to web frameworks, Sinatra defines a simple DSL. For example, `get /items` defines the "/items" URL. When a HTTP-GET request is send to this URL, the corresponding method body is called. This is the controller part of a *Model-View-Controller* pattern, and using views and models in Sinatra is very straightforward too. So, does Sinatra implement a DSL for the total *Model-View-Controller* pattern? We think no, because only the controller part of is realized. But an instance of the Sinatra is an instance of the *Model-View-Controller* pattern.

This answers the first part of our question how patterns and language engineering patterns fit together. In addition to the role of a DSL as an implementation of components from patterns, and as a possible instance of pattern too, we can also relate the abstraction levels of the patterns proposed by BUSCHMANN ET AL. and the language engineering patterns: *Language Integration* provides the framework, the interface to interaction with other programs and languages, *Language Modeling* the design and overall appearance of the language, and *Language Purification* plus *Support* as the idioms of providing high language expressiveness and implementation-details of the DSL. These relationship are shown in ►Figure 7.

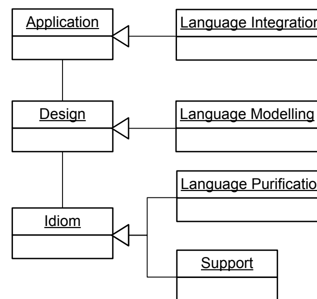

Fig. 7: Abstraction levels of the language engineering patterns

5.3 Feature-Oriented DSL Engineering

Our FOP-DSL rbFeatures [34] [35] can be used to further augment the DSL engineering process. At first we talk about designing a DSL to be feature-enabled right from the start. From the list of process steps, Language Design and Language Implementation are affected. During design phase, one additional artifact needs to be created: The feature model. This model names all features, shows their relationships and constraints. During Language Implementation, we can use the feature model to determine iteration goals. Furthermore, we use feature-containments immediate while programming. This will possibly provide better

[9] http://rack.rubyforge.org/

tests right from the start. In the other case, the DSL already exists, but needs feature-refactoring. The first step is similar - we need a feature model. We then start small iterations which change part of the code and its according tests, and enable features step by step.

But what could govern the decision to have a DSL feature-enabled? We can think of two use-cases. The first case is to provide different variants of the DSL. For example if the DSL is complex because it solves a complex domain. If the features target the *Domain Objects* and *Domain Operations*, we can customize the DSL to only contain a small subset. This can improve using the DSL. The other use case considers more language-internal qualities. Imagine a DSL for transaction control used within a web application. For performance reasons, certain checks are disabled if the frequency of requests exceeds a certain threshold. Having a feature-enabled DSL, we now deactivate this feature, and all currently running and following processes disable the checks. Requests are now processed with the same speed, and once the frequency lowers again, the checks are re-activated. This dynamic-runtime activation would only require an external trigger measuring the frequency, and this would call the feature-related code. As the research continues we can possibly provide more use-cases for dynamic DSL-adaptability in Multi-DSL Applications.

5.4 Language Independence

We suggested a process for engineering internal DSL in a dynamic programming language. At the heart are agile practices with behavior-driven development and patterns. None of these parts is language specific per se, but we choose Ruby as our implementation language. We suggested patterns derived form our work with Ruby. But to what extent can this pattern be generalized to work with other dynamic programming languages?

The work of Dinkelaker and Mezini [12] explains the design of an aspect language for supporting Aspect-Oriented Programming [24]. They use Ruby as the working language, but suggest their results are generalizable to other languages, as long as these languages support four properties: (1) object-oriented constructs, (2) closures, (3) metaprogramming capabilities, and (4) a flexible, non-intrusive syntax. Most important dynamic programming languages support these characteristics, but to a different degree. Especially the required metaprogramming capabilities may differ, and the amount of syntactical freedom for forming expressions.

This leads us to the generalization of the suggested patterns. Language Modeling patterns depended on object-oriented mechanism, and can thus be reused completely. Also, the Language Integration patterns are presented in a general form. Details, such as the available *Hooks* in other languages, may differ, but in general the same ideas can be applied. While these patterns have a good chance of being reused with other dynamic programming languages, the Support and the Language Purification idioms are most likely to fail: Because they use Ruby internals respective work directly on the Ruby syntax, they are hard to apply in other languages.

We conclude that the process can be reused. Engineering DSL with the three phases Domain Design, Language Design and Language Implementation, and using agile development practices including behavior-driven development, is transferable to other languages. Patterns for integrating the DSL into other applications and languages, and for transforming the domain model to a language model, are also usable in other languages. The idioms for Language Purification and Support instead require more work. We think that some of them are usable in other languages too, but this needs to be analyzed. Also, yet undiscovered idioms may reside with other languages and could be added to the pool of patterns and idioms for language engineering.

6 Summary and Future Work

This report proposed a novel engineering process for internal Domain-Specific Languages. By building upon an existing language infrastructure and using an agile process with language statements as the specification, custom DSL can be developed quick and efficient. The paper showed in detail how the agile process works: (1) Collect domain statements and define the domain model, (2) design language syntax, and (3) iterate with language expressions as test specifications. The second part of the report presented patterns for implementing DSL. Patterns play an important role, both because they boost productivity by applying proved solutions to similar problems, and because they ensure further language properties. We distinguished into patterns for Language Modeling and Language Integration, and idioms for Language Purification and Support. Twenty patterns emerged as the result of prior empirical work and the analysis of other existing DSL. In the discussion, we explained how general known software development patterns relate to our language engineering patterns, how to support developing DSL with features, and how our findings can be generalized and used with other dynamic programming languages for language engineering.

The current results must be evaluated with future work. We plan on using the process to build an internal DSL for software configuration management, authorization management, and to further evolve our rbFeatures DSL. At the next stage, we will also research how to use multiple DSL in developing medium-sized web applications. We believe that using multiple existing DSL alongside self-engineered ones greatly improve the development productivity.

Acknowledgements

We thank Maximilian Haupt, Matthias Splieth and Sagar Sunkle for helpful comments and improvements to the paper. We also thank André Zwanziger for a fruitful discussion about patterns which influenced the reports argumentation.

Sebastian Günther works with the Very Large Business Applications Lab, School of Computer Science, at the Otto-von-Guericke University of Magdeburg. The Very Large Business Applications Lab is supported by SAP AG.

References

1. C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language - Town, Buildings, Construction.* Oxford University Press, Oxford, 1977.
2. P. Avergeriou 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. Universitätsverlag Konstanz, 2005.
3. J. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley Publishing, 1996.
5. C. Consel and R. Marlet. Architecturing Software Using A Methodology for Language Development. In *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Berlin, Heidelberg, New York, 1998. Springer.
6. S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain Specific Development with Visual Studio DSL Tools.* Addison-Wesley Professional, Amsterdam, Netherlands, 2007.
7. P. Cooper. 21 Ruby Tricks You Should Be Using In Your Own Code. Webpage, http://www.rubyinside.com/21-ruby-tricks-902.html, last access 11.26.2009, 2008.
8. J. O. Coplien. *Multi-paradigm design for C++.* Addison-Wesley, Boston, San Francisco, et. al., 1999.
9. M. J. E. Cuaresma and N. Koch. Requirements Engineering for Web Applications - A Comparative Study. *Journal of Web Engineering*, 2(3):193–212, 2004.
10. 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.
11. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, San-Franciso et. al., 2000.
12. 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, 2008. ACM.
13. D. Flanagan and Y. Matsumoto. *The Ruby Programming Language.* O-Reilly Media, Sebastopol, 2008.
14. M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, San Francisco et. al., 2003.
15. 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.
16. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering.* Pearson Education, Upper Saddle River, 2003.
17. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley Publishing, Indianapolis, 2004.
18. 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, 2009. Shaker.

19. S. Günther and S. Sunkle. Enabling Feature-Oriented Programming in Ruby. Technical report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, 2009.
20. 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.
21. P. Hudak. Modular Domain Specific Languages and Tools. pages 134–142, 1998.
22. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, New York, 2008. ACM.
23. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modelling - Enabling Full Code Generation*. John Wiley & Sons, Hoboken, USA, 2008.
24. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Berlin, Heidelberg, New York, 1997.
25. F. v. Kutschera. *Sprachphilosophie*. Wilhelm Fink Verlag, München, 2nd edition, 1975.
26. P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.
27. P. Lehmann. *Meta-Datenmanagement in Data-Warehouse-Systemen - Rekonstruierte Fachbegriffe als Grundlage einer konstruktiven, konzeptionellen Modellierung*. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2001.
28. R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Productline Methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GPCE)*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24, Berlin, Heidelberg, New York, 2001. Springer.
29. R. C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, Boston, Indianapolis et. al., 2009.
30. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Survey*, 37(4):316–344, 2005.
31. R. Olsen. *Design Patterns in Ruby*. Addison-Wesley, Upper Saddle River, Boston et. al., 2007.
32. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, 1997. Springer.
33. D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.
34. S. Sunkle, S. Günther, and G. Saake. Representing and Composing First-class Features with FeatureJ. Technical report (Internet) FIN-017-2009, Otto-von-Guericke-Universität Magdeburg, 2009.
35. S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. U. Rahman, G. Saake, and S. Apel. Features as First-class Entities - Toward a Better Representation of Features. *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 27–34, 2008.
36. D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, 2009.
37. A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.

38. J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.