Agile Engineering of Internal Domain-Specific Languages with Dynamic Programming Languages

Sebastian Günther, Maximilian Haupt, Matthias Splieth
School of Computer Science
University of Magdeburg
Germany
Email: sebastian.guenther@ovgu.de, maximilian.haupt@st.ovgu.de, matthias.splieth@st.ovgu.de

***Abstract*—Domain-Specific Languages (DSL) abstract from the domain entities and operations to represent domain knowledge in the form of an executable language. While they solve many of the current software development challenges, related literature claims that DSLs usually have a flaw: The high effort required to implement and use them. However, internal DSLs are developed with less effort because they are built on top of an existing programming language and can use the whole language infrastructure consisting of interpreter, compiler, or editors. This article presents an engineering process for internal DSLs. An agile process leads from analysis to design and implementation. Expressions and language capabilities are implemented using tests and a set of patterns, which provide reusable knowledge how to properly structure and design the DSL implementation. As a case study, we show how to implement a software product line configuration DSL using Ruby and Python as host languages. In summary, the proposed process and patterns facilitate the successful planning and developing of internal DSLs using dynamic programming languages as the host.**

***Keywords*-domain-specific languages**

I. Introduction

Domain-Specific Languages (DSL) are tailored towards a specific application area [20]. They use domain-specific notations and abstractions [29] to represent domain knowledge in a precise and reusable form. Benefits of DSLs comprise efficient code reuse, increased productivity and significant error reduction [10], [16]. DSLs cover diverse application areas, like healthcare systems [24], financial products [1], or web applications [12]. A good overview of all research topics in the DSL field can be found in [29].

DSLs are differentiated into their appearance (textual vs. graphical [6]), their origin (internal vs. external [10]) and their implementation (interpreter, compiler/generator, preprocessor, embedding, extensible compiler/interpreter, commercial of the shelf, and hybrid [23]). The origin is a crucial characteristic. *External DSL* allow to define the language's syntax and semantic freely, at the cost of providing code-generators and editors. *Internal DSL* instead are built on top of an existing programming language [23], which is also called the host language. Several modifications and extensions are applied to the host language in order to represent the domain.

External DSLs require considerable investments, as new languages, tools and techniques have to be learned. This investment is only taken when the DSL has a profound impact on development productivity [23]. In contrast, internal DSLs require less effort: Developers can reuse the existing language infrastructure, consisting of compiler, interpreter and editor. And since each DSL expression is also an expression of the host language, the integration with other DSLs, frameworks, and application written in the same language is simplified. In our view, these benefits prevail possible constraints.

Our focus is internal DSLs – and we mean this specific type whenever we speak of DSL in the following. In prior work, we implemented several DSLs with the *Ruby* programming language. One DSL expresses the configuration of software product lines with their structure and constraints [18]. Another one enables to use feature-oriented programming in Ruby by implementing features as first-class entities inside a program [19]. We reflected and saw that we used several steps in an iterative nature to analyze, design and implement the DSL. We also used patterns to address DSL-engineering specific problems.

This paper's contribution is the summary of our own DSL development experience: An agile DSL-engineering process using patterns. Iterations combining analysis, design, and implementation extend the DSL with new expressions or provide enhanced language capabilities. Constant refactoring keeps the code base minimal and extensible. Patterns – meaningfully structured to support several DSL-engineering concerns – provide reusable knowledge how to properly structure and design the DSL's implementation. The process utilizes dynamic programming languages, because they require less amount of code for programs [26], which increases the languages readability, and their support for metaprogramming, which has been reported to be beneficial in DSL-engineering [9][11]. Using this process in the context of application development allows DSLs to become by-products that immediately increase the productivity for the current project and are eligible for future reuse.

Section 2 explains the DSL-engineering process. We start with explaining the general properties of our process, then explain the phases, and finally the patterns. Patterns address the three DSL-engineering concerns of *Language Modeling* (provide executable form of the domain model), *Language Integration* (integrate the DSL into the application framework and with other DSLs), and *Language Purification* (improve language readability). Patterns will be only summarized: a detailed explanation can be found in the accompanying technical report [17]). In Section 3 we present how the process and the patterns are applied. We develop a DSL for the configuration of software product lines, thereby using Ruby and Python as host languages and showing the implementation side-by-side. Afterwards, Section 4 discusses related work on DSL-engineering process and implementation mechanisms. Finally, Section 5 concludes this paper. We apply these formats: *keywords*, *features*, and `DSL expressions`.

II. Agile Engineering of Internal Domain Specific Languages

The goal of our internal DSL-engineering process is to use as much existing knowledge as possible. Therefore, the process uses steps that are common to application development in general. In the

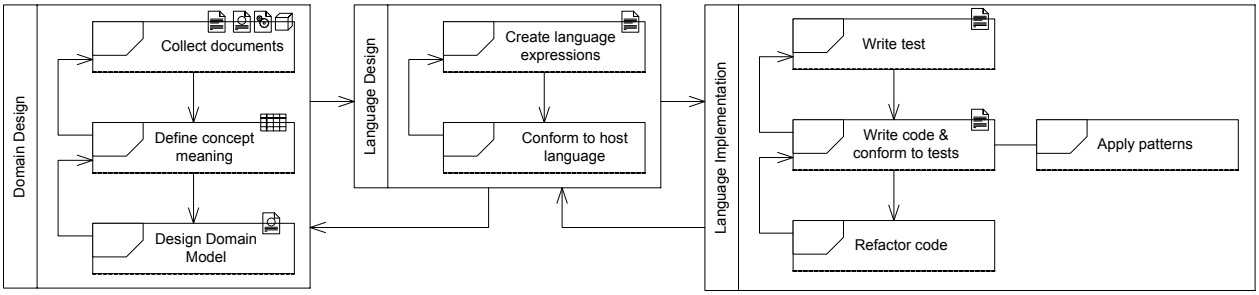

Figure 1: The Agile DSL-Engineering Process

domain design phase, the static and the dynamic domain model are developed. The model expresses entities, attributes, relationships, and operations. Then, the *language design* phase creates DSL expressions that represent this domain model. Finally, the expressions are implemented in the *language implementation* phase. Several pattern that support common DSL-engineering problems are used in this phase. The development phases and concrete steps are shown in Figure 1.

The agile nature of the process follows the insight that successive iterations result in a better domain understanding. Iterations typically take a small amount of domain knowledge, which either results in an extension of the DSL with novel language constructs or improves the DSL's capabilities. The result of each iteration is a tested and running DSL implementation, typically a set of objects and methods. This implementation is then refactored and serves as the starting point for a new iteration.

Integrating this process in the course of general application development is straightforward. After the initial analysis, parts of the application can be identified to be expressed with the help of DSLs. Then, from requirements to implementation, the agile process continuously develops the DSL and the specific part of the application too. This enables DSLs to become by-products of application development.

The next sections detail the process phases and the patterns. The process is the result of several DSLs we developed, among them one for modeling software product lines [18] and one for feature-oriented programming [19]. For finding patterns and idioms, we did not stick to our empirical gained knowledge alone, but also studied the open-source Ruby DSL HAML (http://haml-lang.com/; HTML), SASS (http://sass-lang.com/; CSS), DataMapper (http://datamapper.org/; database connector), and Sinatra (http://sinatrarb.com/; web application framework). For Python we used Bottle (http://bottle.paws.de/; web application framework) and SQLAlchemy (http://sqlalchemy.com/; database connector). Further analyzed literature is [25][13][28].

A. Domain Design

In the beginning of DSL-engineering, the first goal is to develop a deep-founded understanding of the domain. Various handbooks, documentation, systems and general stakeholder expressions are collected: this is called *domain material*. The material is studied to produce either formal or informal expressions about the domain. One form is to use variability and commonality analysis and collect statements in natural-language about the domain [9]. Other forms are domain engineering techniques like FODA (Feature-Oriented Domain Analysis) [10]. If only experts possess the required knowledge, creative techniques like brainstorming or more formal questionnaires (checklists etc.) [8] are applicable. Special attention should be given to seemingly contradicting statements – they point at misunderstandings of the domain that need to be resolved. A profound understanding (not necessarily a "complete" specification!) of the domain guards against undesired changes in later iterations.

The collected statements contain singular and compound expressions about the concepts and the relationships. The gained knowledge is then refined to the static *domain model*, consisting of the concepts, attributes and their relationships. As well as the static domain model, its dynamic counterpart is important too, where domain concepts interact according to *domain operations*. Concrete model representations are specific to the concrete application development process. One suggestion is to use the UML class diagram for the static structure of entities, attributes and relationships, and the UML state diagram to represent the different status of the domain. When we speak of domain model in the following, we mean both its static and dynamic part.

B. Language Design

In the language design phase, we develop the syntax for the DSL. Both the static and the dynamic domain model are considered, so that all domain concepts, attributes, relationships, and operations can be expressed. DSL 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. An useful metaphor is that of a *language game*. The philosopher Wittgenstein used language games to determine the grammatical correctness of expressions [21]. Such language games can be used with a compiler or interpreter. If a DSL expression raises only semantic errors, then it is syntactical valid according to the host language. The second approach works vice versa – taking host language expressions, and simplifying them to increase the language's readability. Here, developers use their language knowledge to simplify relationship expressions, improve readability of passing arguments to methods, and generally remove domain-foreign symbols and token. The important point is to provide a high readability. It is naturally to disambiguate method parameters, to structure expressions according to the natural hierarchy of the domain entities, and more.

C. Language Implementation

In language implementation, a form of behavior-driven development is used: We first provide a test, and then its implementation. Having a set of DSL expressions available, first iterations typically use *example expressions as test cases* to build an implementation that provides the semantics for the used objects and operations. In Ruby, the *RSpec* (http://rspec.info) library can be used, and in Python, we can test the result of DSL expressions with *assert statements*. In later development stages, tests are written for extending the language capabilities or to cover errors. After passing all tests, we refactor our code to provide a minimal and sufficient DSL implementation.

The implementation is supported by patterns. In computer science, a pattern names and explains "... an important and recurring design" [15]. Patterns are a way to record mature and proven design structures [7], and furthermore establish a vocabulary to describe solutions [14]. The focus of the shortly presented patterns is to provide a clear expression of the domain in terms of its concepts and operations, to have a high readability of the language, and to integrate the DSL with other DSLs and frameworks. These concerns are the core points that provide readability and maintainability for the application part written with the DSL. Other Concerns like domain-specific errors or optimization [23], and constraints regarding the language's usage, are not covered here. Several patterns have a close relationship to or are even similar to existing patterns. The difference of patterns used for application development and for DSL-engineering lies in the kind of abstraction they construct upon, which is a language in our case.

The patterns are grouped according to the following DSL-engineering concerns:

- *Language Modeling* – All concepts, attributes, and operations in the domain form the vocabulary. Naturally, this should form the basic structure of the DSL too. In object-oriented programming languages, we can use modules, classes, instances, and methods for this purpose.
- *Language Integration* – Using the DSL in isolation limits its potential. The real value lies in the integration with other domain-specific languages, libraries, and frameworks. The language integration patterns help to decouple and structure the parts of the DSL so they can work together with other components easily, or they show how expressions can be integrated.
- *Language Purification* – Certain syntactical constraints of the used host language can be a burden to the DSL. Language Purification is the task of eliminating non-domain relevant symbols and tokens by providing syntactical improvements or alternatives, and thus to improve language readability.

All patterns are explained in Table I. The table summarizes the patterns with their name, a short description, and indicate whether they are available in Ruby or Python (a "+" means it is available, a "(+)" means limited availability, and a "-" means not available). The accompanying technical report [17] details these patterns and shows implementation examples.

III. Example: The Software Product Line Configuration Language

Software Product Lines address the challenge of structuring and systematically reuse software by providing a set of valuable production assets [10]. Assets are documentation, configuration, source code, libraries, handbooks and much more. Structurally, a "product line is a group of products sharing a common, managed set of features" [30], where features describe modularized core functionality [4].

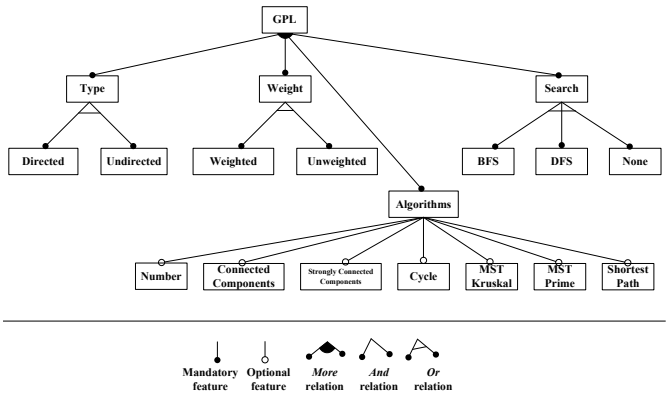

Figure 2: The Graph Product Line

This section explains how we used our DSL-engineering process to implement a DSL for configuring software product lines. Ruby and Python are used as the host languages. The goal of the DSL is to provide an internal model of a product line. The model can be used to test the programs configuration at runtime. We show the processes with side-by-side explanation of the domain analysis, language design and implementation. We do not show the iterations per phase, but summarize the results of several iterations in the explanation.

A. Domain Analysis

We began our study with various scientific work on software product lines and feature modeling [30][10][22]. This material formed our initial understanding of the domain. As the background example, we choose the often cited graph product line [22].

The graph product line is shown as a feature diagram in Figure 2. At the top of the graph, we see *GPL* – the root node, representing the graph product line itself. At the next layer, four features are defined. *Type*, *Weight*, *Algorithms*, *Search*. Each of them is a mandatory feature that has to be included inside the product line. *Weight* determines whether the edges of the graph have a weight or not – thus, the corresponding features *Weighted* and *Unweighted* are alternatives. The optional *Algorithms* determine the available graph operations.

With the addition of these constraints, we can formulate our domain understanding. We distinguish into the following entities and operations.

- *Feature* – An entity realizing a set of functionality important to a stakeholder.
- *Relationships* – Features can have `subfeatures`, and they take a specific place inside the feature tree (`root`, `node`, `leaf`).
- *Constraints* – Features impose constraints upon other features, which we name `require`. The constraints are to strongly require a specific feature (`is`), to select `one` or `more` from a list, or to provide `any` other feature.

Table I: Language Modeling Patterns

Type	Pattern	Description	Ruby	Python
Modeling	Command	*A quick way to provide objects that execute domain-specific operations.* The command pattern, as originally introduced in [15], defines an abstract "Command" class. Subclasses overwrite the "execute" method with the domain-specific operation. This pattern is good to quickly implement functionality, but does not provide adequate support for complex domain models.	+	+
	Domain Objects	*Entities and their properties are represented as objects with attributes.* Using object-oriented mechanisms, entities are expressed as modules, classes and instance objects – dependent on the required mechanisms how to combine and extend the objects. Working with domain objects directly in expressions is an indicator for readability.	+	+
	Domain Operations	*Provide methods and functions to express status changes in the domain or relate domain objects to each other.* Representing domain operations as functions that are flexible in terms of receiving and parsing arguments help to express complex domain operations.	+	+
Integration	Internal Interpreter	*A global interpreter object executes both internal and external DSLs.* While the internal DSL directly manipulates its language model in terms of objects and attributes, statements of the external DSL need to be parsed and analyzed for execution.	+	+
	Hooks	*Use host language specific or self created hooks for intercepting DSL execution.* 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. DSLs can use both hooks to customize the expression execution, like calling statements of another DSL.	+	+
	Language Modules	*Provide parts of a DSL as reusable modules.* Modules serve as containers for functionality, which is copied to different entities. By directly modifying the module or by combination of modules from different DSLs, integrated language expressions describe the interaction of application parts.	+	+
Purification	Keyword Arguments	*Use named arguments when calling methods to explicitly state their meaning.* Method calls with more then two parameters require the user to know their type and order. Arguments indexed by keywords express the meaning of arguments and can further be used to form natural language like sentences.	+	+
	Block Scope	*Provide a clear context for evaluating statements or stack hierarchical information.* Ruby supports closures and anonymous code blocks. Code is specified in one place, but can be executed in any other. This mechanism can be used to provide (i) a clear execution context as expressions have an explicit place, (ii) seamless method extensions by passing closures to other functions that are used in combination, and (iii) to express structured data with a hierarchical layout. Block Scope does not work with Python, since method calls are always fully-qualified.	+	-
	Method Chaining	*Statements of chained methods mirror natural language expressions.* Instead of using one-line commands typical for programming languages, chain method calls that always manipulate the same entity. This can improve the language's readability, but requires semantic changes within the methods.	+	+
	Superscope	*Use strings and symbols to transcend execution scope.* Using the classic *Proxy* pattern [15], symbols and strings can be used to reference entities in another scope, thus decoupling DSL expressions from the application's implementation.	+	+
	Parentheses Cleaning	*Eliminate parentheses around method calls to improve language readability.* Parentheses are a necessity of most programming languages, but usually do not carry semantic information. In most cases, Ruby allows to drop parentheses. Bun in Python, they are required.	+	-
	Boolean Language	*Use natural language for boolean operation.* The standard boolean operators *and*, *or*, and *not* are part of Ruby and Python language. They and can be used instead of their symbolic representation.	+	+
	Operator Redefinition	*Redefine language symbols to be used in DSL expressions.* Symbols for addition, subtraction and more are a natural way to express relationships among domain objects. Many symbols in Ruby and Python are actually (re)definable operations on objects. They can be changed for specific objects only, or those defined in a specific namespace.	+	+
	Custom Return Objects	*Return multiple values with the simplest data store – a custom object.* Parsing multiple return values typically requires structural knowledge on the receiver side. If we put the data inside an object, and access its values directly, the languages readability is improved.	+	+
	Aliasing	*Change existing methods to have a more domain-specific name.* By changing the names of methods and objects, even built-in types can look like domain entities. While Ruby allows to manipulate core classes, Python demands to build subclasses first, which limits this patterns applicability. As an alternative, the classical *Proxy* pattern can be used too.	+	(+)
	Seamless Constructor	*Create instances without using the new operator.* In Ruby, the "new" operator expresses the intent to initialize a new instance of any class. Some DSLs may need new objects, but don't want to use the "new" operator at all. Ruby allows redefining the constant for the class as a method of the same name, which in its body calls "new". In Python, instances are created seamlessly per default.	+	+

- *Product Line* – A set of features, relationships, and constraints. A product line is `valid` if all constraints are satisfied.
- *Product Variant* – A concrete configuration that is either `valid` or not valid according to its product line.

B. Language Design and Implementation

Because we needed to experiment with Ruby and Python, Language Design and Implementation was an interwoven process. To keep the explanation focused, we will only present how the `Feature` entity was implemented. The final expressions to configure features are shown in Figure 3 – the next paragraphs detail their meaning.

```
1 gpl = Feature.configure do
2   name :GPL
3   root
4   subfeatures :Type, :Weight, :Search, :Algorithms
5   requires :GPL => "all :Type, :Weight, :Search,
        :Algorithms"
6 end
```

A) Ruby

```
1 with Feature() as gpl:
2     gpl.name = 'GPL'
3     gpl.root()
4     gpl.subfeatures("Type", "Weight", "Search",
          "Algorithms")
5     gpl.requires("GPL", "all Type, Weight, Search,
          Algorithms")
```

B) Python

Figure 3: Defining Features with the DSLs based on Ruby and Python

We started with the design of feature definitions. We wanted to have a syntax that reads naturally and explains the feature's name, position in the feature tree, and constraints. Many Ruby programs use `do...end` blocks to express code within a named context – this is the *Block Scope* pattern. This syntax is appealing because it reads naturally. We designed the first Ruby expression to be `Feature.configure do...end` (Line 1). The expressions in this block are executed in the contexts of its receiver – an instance of `Feature` in this case. Python has a similar concept called *context manager*, which results in the expression `with Feature()as gpl` (Line 1). But inside the context manager, method calls require a fully-qualified name, so the `gpl` object has to be put in front of every statement inside the context manager's body.

The next part was to express the feature properties. Initially, we just used basic assignments, like `name = "GPL"`. But this notation is typical for programming languages. We wanted to have a notation that expresses the properties as statements. Therefore, the next iteration evolved the language to use method calls. Ruby allows to drop the parentheses with the *Parentheses Cleaning* pattern, so this statement could be rewritten to `name "GPL"` – in Python, parentheses are required, which makes this statement `gpl.name("GPL")` (Line 2). We also used this notation to express the position a feature has inside the feature tree. For Ruby, this statement is `root`, and in Python `gpl.root()` (Line 3).

The definition of subfeatures evolved through two versions. The first version required to use concrete objects. However, in order to modularize feature declaration and to decouple the declaration order, the next iteration introduced the *Superscope* pattern to resolve the symbols to point to the real objects at execution time. In Ruby, this is expressed as `subfeatures :Type, :Weight, :Search, :Algorithms`, and in Python `gpl.subfeatures("Type", "Weight", "Search", "Algorithms")` (Line 4).

The constraints were a challenge. We imagined an expression with the feature responsible for the constraint to the left, a keyword specifying what kind of constraints, and then one or more features to the right. The solution is the *Keyword Arguments* pattern. Although we could have used the *Block Scope* pattern again, we wanted to express constraints in one line only, which is arguably more readable for multiple constraints. We used a string to express the constraints, which is actually a method call executed with the `eval` metaprogramming method. This works both in Python and in Ruby. Thus, the last expression to configure a feature is in Ruby `requires :GPL => "all␣:Type,␣:Weight,␣:Search,␣:Algorithms"` respective in Python `gpl.requires("GPL", "all␣Type,␣Weight,␣Search,␣Algorithms")` (Line 5).

As we can see, the steps and the applied patterns are very similar. However, the final DSL has syntactical differences due to one important construct: Rubys anonymous code blocks allow eliminating the caller, while the caller must always be expressed in Python. Because the caller and parentheses can be left out, the resulting Ruby-based DSL is more readable. This concludes our case study.

IV. Discussion and Related Work

Comparing our approach with related work, we see that both the *process* as well as the concrete *mechanisms* to engineer DSLs differ. Both are explained in the following.

A. Process

Depending on the DSL type, different processes can be found. For internal DSL, a process using commonality and variability analysis to capture the domain in plain English is described in [9]. Expressing the domain model in a class diagram like form is shown in [11]. From the class diagram, the language's syntax using EBNF is defined, and then all language elements are implemented. For external DSL, some publications explain complete development environments [2][3]. The environments require to specific the DSL's syntax and semantics using formal expressions. A tool-independent approach is shown in [5], which suggests to use commonality and variability analysis to elaborate language requirements, objects and operations, and notations. Independent of the DSL type, [23] describes are general process. It has similar phases than our process except for two additional phases. First, the decision phase analyzes whether a DSL should be implemented at all. Two criteria are supporting the concerns to improve software economics and to enable software development by users with less amount of knowledge. Second, the deployment phase – although only briefly mentioned – targets the accompanying user training phase or developing tool-support for using the DSL.

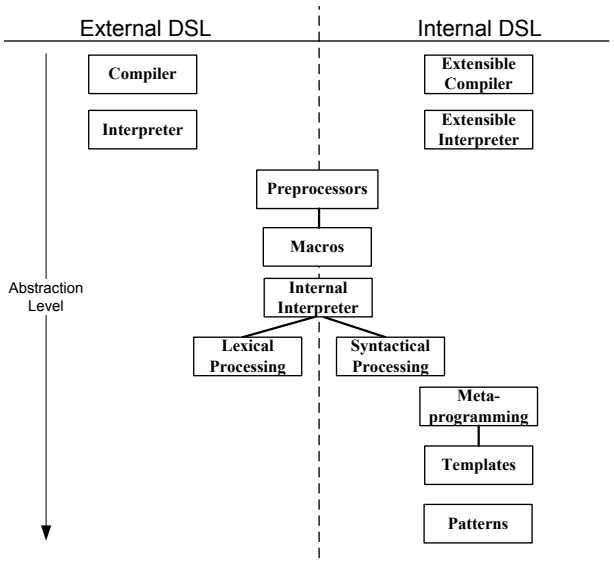

Figure 4: DSL-Engineering Mechanisms

Our approach contrasts from these explanations. At first, we do not prescribe what specific technique should be used to understand the domain. We argue that the availability of proven design techniques strengthens the DSL-engineering success and utilization. Second, the process requires fewer investments then other approaches. The only investment is into understanding the used languages and its metaprogramming mechanisms. Ultimately, this improves developers productivity with the whole language and impacts the application as well. Third, we emphasize the agile nature. Each iteration produces a working DSL – ready to be used with the current development phase. We are certain that the agile approach has long term benefits on the languages design and implementation. And fourth, since the DSL stays within the language, it has better portability to other interpreters that implement the full language specification, and it is open to integration with arbitrary other DSLs, applications, and frameworks. In summary, if developers learn to see DSL-engineering as a means to solving application development problems, DSLs can become by-product of software development: Increasing the productivity of the current development, and at the same time suitable for future reuse within other projects.

B. Mechanisms

If we consider all mechanisms mentioned in [5][10][27][23][9][11] and extend it with our findings regarding the use of patterns, then we see the general available mechanisms of how to engineer DSLs in Figure 4. From left to right, we see the available mechanisms for each DSL type. Mechanisms depicted in the middle are available for both types. From top to bottom, we denote the increasing abstraction level.

In Figure 4, we find extensible compilers and interpreters on the far end of abstraction. They require detailed knowledge of the complete interpretation and translation phase, of the used implementation language, and of the structural constraints extensions need to regard. On the other end, patterns and metaprogramming stay at the abstraction level of the utilized languages: Developers just need to understand how to change the internal meaning and behavior of their objects through host language expressions. This kind of modification is more abstract then modifying the interpreter. Furthermore, experienced programmers usually possess knowledge about pattern and metaprogramming. Therefore, the great benefit of our approach is that the required knowledge – and with it the required time – to implement the DSL is very modest in comparison to other DSL-engineering mechanisms.

Considering related work, only some authors try to structure the host language's modification. They usually speak of techniques, and mix language characteristics with implementation decisions. Our works helps to pinpoint the changes in the form of patterns, and the provided pattern language can be used to communicate and plan the solution. We can even explain DSL implementation in yet not studied host languages. For example, the approach in [9] uses Ruby, the mechanisms Metaprogramming, Internal Interpreter with Syntactical Processing, and the patterns *Domain Objects*, *Domain Operations*, *Parentheses Cleaning*, and *Block Scope*. A DSL using the Groovy programming language is shown in [11]. Clearly, the patterns *Domain Objects*, *Domain Operations*, and *Block Scope* can be seen – although this particular programming language has not been regarded by our process yet. The seaside framework, implemented in Smalltalk, also uses *Domain Operations* and *Block Scope* to implement a HTML DSL [12].

V. Conclusion and Future Work

We conclude that internal DSL can be developed effectively using our suggested process and the patterns. The agile nature of the process has several benefits: (1) The initial domain understanding is continuously refined to match the evolving requirements, (2) the result of each iteration is a complete and writable DSL, (3) the DSL development can be integrated with the application development, producing the DSL as a "by-product". The patterns are an important cornerstone too. They record the development knowledge and provide solutions to common domain-engineering problems. Furthermore, they also provide a pattern language that can be used to communicate or plan the implementation. And since no new languages or tools are required, developers just need to learn and apply the patterns in the context of their existing programming knowledge and can start to agile implement DSLs.

We see two areas for future work. The first area is to analyze existing DSL and to experiment with other dynamic programming languages to refine and extend the pattern catalog. The second area is to detail how the host language influences the DSL's design and implementation.

Acknowledgements

We thank Christian Kästner and the anonymous reviewers for helpful comments on earlier drafts of this paper. Sebastian Günther works with the Very Large Business Applications Lab, School of Computer Science, at the Otto-von-Guericke-Universität Magdeburg. The Very Large Business Applications Lab is supported by SAP AG. Maximilian Haupt and Matthias Splieth are master degree students with the School of Computer Science.

References

[1] 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. IEEE, 1995.

[2] R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.

[3] R. A. Ballance, S. L. Graham, and M. L. V. De Vanter. The Pan Language-Based Editing System For Integrated Development Environments. *ACM SIGSOFT Software Engineering Notes*, 15(6):77–93, 1990.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society, 2003.

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

[8] M. J. E. Cuaresma and N. Koch. Requirements Engineering for Web Applications - A Comparative Study. *Journal of Web Engineering*, 2(3):193–212, 2004.

[9] 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.

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

[11] 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.

[12] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A Flexible Environment for Building Dynamic Web Applications. *IEEE Software*, 24(5):56–63, 2007.

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

[17] S. Günther. Agile DSL-Engineering and Patterns in Ruby. Technical report (Internet) FIN-018-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[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. 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.

[20] P. Hudak. Modular Domain Specific Languages and Tools. In P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, pages 134–142, 1998.

[21] F. v. Kutschera. *Sprachphilosophie*. Wilhelm Fink Verlag, München, 2nd edition, 1975.

[22] 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.

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

[24] 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, 2007. ACM.

[25] R. Olsen. *Design Patterns in Ruby*. Addison-Wesley, Upper Saddle River, Boston et al., 2007.

[26] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 21(3):23–30, 1998.

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

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

[29] A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.

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