Engineering Domain-Specific Languages with Ruby

Sebastian Günther
Otto-von-Guericke-Universität Magdeburg

Abstract: In the current form, Software Engineering is an integration task. Different types of languages, technologies and ultimately domains must be brought together in a compound form. A fundamental difficulty is how to express the parts of an application in such multi-language developments. Entities of the domain have representations in different, incompatible languages along the applications layer. Naturally, changing the concepts requires updating all representations. As syntax and semantics of the languages differs, such changes are difficult to implement. We argue that such complexity can be remedied with the help of domain specific languages. In our case, we consider internal domain specific languages, which are based on an existing programming language. The following report details our motivation, research steps and current findings.

Keywords: Domain-Specific Langauges, Dynamic Programming Languages, Feature-Oriented Programming

1 Motivation

Software has an inherent complexity. Since the advent of software engineering with the Nato conference in 1968, the question of how to cleanly modularize software into its various concerns is an ongoing question [Apel07]. Today's challenges involve not only multiple requirements and different domains that software must consider.

Research discovered a fundamental problem which hinders the effective treatment of multiple domains. The *tyranny of the dominant decomposition* forces developers to decompose software along one dimension only [Tarr+99]. This leads to several software defects, such as tangled and bloated code [Kicz+97] and structural mismatches of requirements and programs because they can not be mapped one to one. Solution suggested to the code-tangling problems are concepts like Coplien's ideas on multi-paradigm design [Copl00], Kiczales et al. about Aspect-Oriented Programming [Kicz+97] and Prehofer's Feature-Oriented Programming [Preh97].

Another problem we ourselves discovered during research is the *language problem* ocuring when using multiple language in software development. In the evolution of Software Engineering to its current form, many new domains were discovered: Databases, network communication, and web pages to name a few. Typically, each domain developed a specific language which expresses the concepts and operations of the domain. Today's applications, like web applications, require separate languages for accessing databases (SQL), calling external services (XML), presenting to the user (HTML, CSS) and to express the application domain (Java, Python, Ruby). Since the languages formulated their own syntax and semantic, they are incompatible to each other. This leads to defects like domain concept scattering (concepts of the domain have different representations in all languages) and requirements diffusion (central requirements are spread over different artifact of different languages). Because of this, implementation, testing and debugging is more complex then it needs to be. Understanding such applications is also very difficult.

We argue that the usage of Domain-Specific Languages both aid with the multi-domain and the multi-language problem. Domain-Specific Languages, or shorthand DSL, are tailored towards a specific application area [Huda98] and use suitable notation and abstraction mechanisms to represent domain-knowledge and concepts in a precise form [Deur+00]. Our vision is that both the domains and the languages used in development are abstracted with domain-specific languages based on one common programming language. If carefully crafted, such languages can be used interchangeable, and allow to use the same concept in different expressions. This simplifies specifying, implementing, testing and maintaining the application tremendously.

The research goal is to develop a process for engineering Domain Specific Languages and a process to utilize multiple DSL effectively when implementing applications.

2 Domain-Specific Language and Language Engineering

Domain-Specific Languages, or shorthand DSL, are tailored for a specific application area. They use suitable notation and abstraction to represent domain-knowledge and concepts in a precise form [Deur+00]. Because the domain-knowledge is put inside the language, programmers using the DSL will form a better understanding of the domain. DSL bring following advantages: increased productivity, efficient code-reuse, reduction of errors and the focus on the solution space [CzEi00] [Gree+04].

Historically, DSL were mentioned in science as early as 1966 with Landin's seminal paper on "*little languages*" [Land66]. Since then, they have evolved to a fairly common practice in software engineering: From early examples in video processing [Thie+97], financial products [Arno+95], and signaling installations for rails [Groo+95], to modern telephone support [Latr+07], healthcare systems [MuCl07], and web applications [Viss08].

DSL can be distinguished according to their appearance (textual vs. graphical [Cook+07]), their origin (internal vs. external [CzEi00]), and their implementation (interpreter, preprocessor, hybrid [Mern+05]). Different tools and accordant processes have been proposed: The Microsoft Visual Studio [Cook+07] and Metacase [KeTo08] aim at developing DSLs graphically, while textual DSLs can be developed with the ANTLR Toolkit [Tarr07].

Despite their different forms, the question how to develop Domain-Specific Languages is also important. Development processes for DSL are only infrequently discussed, like in [CoMa98] [Spin01] [Mern+05]. Simplifying this literature and especially regarding [Cook+07] [KeTo08], the systematic development of a DSL is usually performed in the following steps: (1) analyze the domain and create a domain model. (2) Create language requirements by taking the concrete syntax (tailored towards end-users), the DSL type (internal vs. external) and overall code-generation criteria (host language, surrounding language framework) into account. (3) Implement the necessary generators. (4) Generate the application code, check for the correctness of code transformations and use the code in production.

3 Research Scope and Purpose

The goal of the research is to remedy the two problems of software engineering: Tyranny of the dominant decomposition and the language problem. Our central hypothesis is as follows:

By building internal domain-specific languages with a coherent base language for multi-domain and multi-language development scenarios, the whole application can be analyzed, specified, implemented, and tested at the same language level.

Furthermore, we want to show that building a set of DSL for an application leads to a seamless integration of the different software development phases. We will prove the hypothesis along the following steps.

3.1 Foundation and Language Selection

Various aspects of DSL, DSL Engineering, Software Engineering and basics of Programming Languages converge to the formal background of the thesis. Central question was what constitutes a DSL and what role it plays in the context of Software Engineering in general. Of special interest were programming paradigms like Aspect-Oriented Programming [Kicz+97] and Feature-Oriented Programming [Preh97]. They help to modularize programs by focusing developers on one functionality at a time.

The other point was to select a suitable base language. In computer science, many different types of languages exist: General purpose languages (C, C++, Java), scripting languages (PHP, Python, Ruby), markup languages (XML, HTML), query languages (XSLT, SQL), and declarative languages (CSS, XML-Schema). For the host language, only Turing-complete programming languages are amenable. Initial research compared general purpose languages and scripting languages. We concluded that scripting languages are the best base because of their dynamic nature, metaprogramming facilities and the syntactical simplification they provide. Comparing Python, Groovy and Ruby in terms of syntactical modification, types and readability, finally lead to selecting Ruby as the base language.

3.2 Practical Language Engineering

Ruby is a clear, yet complex dynamic programming language. To further support and evaluate the theoretical selection, we began to practically design DSL and learned the fine mechanisms of Ruby. In succession, we created a DSL for the configuration of software product lines and for enabling feature-oriented programming in Ruby. These examples are explained in detail in the results section.

3.3 DSL Engineering Analysis

Selecting Ruby was proved as the correct choice. Both the first-hand experiences and ongoing research into web applications, especially written with the Rails framework, shows how concise DSLs can be implemented and used in application development. The next step is to refine the experiences with a detailed study of other DSL. For this, we select a number of DSL from the Rails framework. We analyze the design (the syntax) of the languages and their implementation (the semantics archived with metaprogramming mechanisms). The results will be a catalogue of DSL patterns and a guideline how to use metaprogramming for implementing DSL.

3.4 Language Construction and Application Construction Process

Early work already suggested two basic processes. *The Language Construction Process* provides a step-by-step explanation how to design a DSL. It begins with capturing informal expression of the domain, formalized the domain concepts and operations, designed a syntax and implemented the language. *The Application Construction Process* took a step back from traditional modularization concepts, and suggested to analyze the different domains in which a application operates. By designing the domain interfaces as language interactions, more concise development can be archived. However, both processes focused too much in the role of the domain and neglected the technical side of working. Thus, the process must be redefined with recent experiences in DSL implementation.

3.5 Multi-DSL Application

With the refined process, the catalogue of DSL patterns, profound experience in metaprogramming and overall experience with web application development in Rails, we can begin the last step of the dissertation. We will design a medium-sized web application as the case study which validates the applicability of the processes. The exact topic and domain of the application is yet to be specified.

4 Results

We details the two DSL developed for configuring Software Product Lines and for enabling Feature-Oriented Programming with Ruby. For better understanding, we apply following text formattings: *keywords*, FEATURES (for Feature-Oriented Programming) and language constructs (both Ruby and DSL terms).

4.1 Software Product Line Configuration Language

Software Product Lines are a paradigm that address the challenge of structuring and systematically reusing software by providing a set of valuable production assets [CzEi00]. Assets have the form of documentation, configuration, source code, libraries and more. Typically, a "product line is a group of products sharing a common, managed set of features" [Whit96], where features describe modularized core functionality of a software [Bato+04].

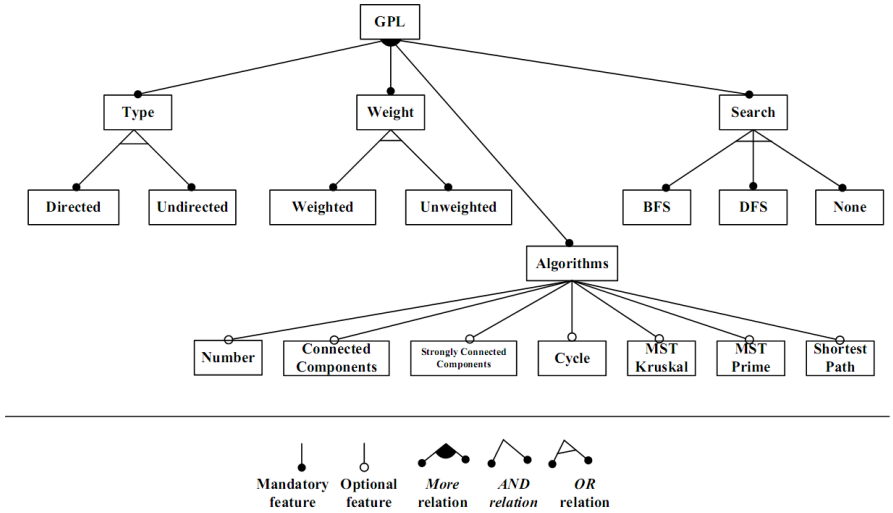

Figure 1: The Graph Product Line

The Software Product Line Configuration Language, or shorthand SPLCL, addresses the definition of a product line form a modeling perspective. A so called feature tree structures the features and their relationships to other features. SPLCL realizes the feature tree as feature entities and their relationships. Features are of type root, node and leaf. Relationships use the keywords all, any, one, more and is to relate their selection in the tree with the existence or choice from other features.

For explaining the DSL, we use the example Graph Product Line (GPL) from [LoBa01]. In Figure 1, we see the hierarchy of features and their relationships. As can be seen, the product line differentiates the type and weight of a graph, provides search algorithms, and implements numerous algorithms.

The syntax in SPLCL for configuring features is explained with following Figure 2. Line 1 shows the definition of a feature. We use the configure method and a do...end block for configuring the feature. Line 2 defines that GPL is the name of the feature, and line 3 makes it the root feature of the whole product line. The next line 4 lists all subfeatures of GPL, and finally in line 5 we see the definition of the features relationship. The invoking of the root feature requires that all the features, TYPE, WEIGHT, SEARCH and ALGORITHMS. As can be seen, lines 2 to 5 directly express the domain, without typical programming language syntax. This is a good example for the types of DSL which can be archived with Ruby.

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

Figure 2: Syntax for Configuring a Feature

Once all features are created, the next step is to configure the ProductLine. It receives an description and adds features with add_feature. The product line has built in logic for checking if only one root exists, if all named features in the sub feature relationships are included, if all features are connected to each other and if their type corresponds to their position. See Figure 3.

```
ProductLine.configure do
  description "The complete GraphPL as a working example for Ruby FOP"
  add_feature gpl_feature
  add_feature(type_feature)
  add_feature(weight_feature)
end
```

Figure 3: Defining a Prodcut Line

The final step is to create a ProductVariant, as showsn in Figure 4. Again, it receives a description and a ProductLine as a base. The ProductLine is checked for validity. Features are added by using activate_feature. Validity checks see if all constraints hold. Figure 4 depicts that a basic variant with the features GPL, TYPE, WEIGHT, SEARCH and ALGORTITHMS is created.

```
ProductVariant.configure(graph_product_line) do
  description "Basic variant with algorithms Cycle and Numbers only"
  activate_feature :GPL
  activate_feature :Type
  activate_feature :Weight
  activate_feature :Search
  activate_feature :Algorithms
end
```

Figure 4: Defining a Prodcut Variant

4.2 Feature-Oriented Programming Language

While SPLCL helps in modelling a product line, it does not help in actual implementation. For this, we developed *rbFeatures*, a DSL for enabling Feature-Oriented Programming in Ruby.

Typical approaches to FOP have several drawbacks as explained in [MeOs04] [Kaes+08] [Sunk+08]. We analyzed the problems and concluded that realizing features as first-class entities remedies many problems [Sunk+08]. First-class entities means that features themselves are objects of the languages which can be created and used in arbitrary expressions. rbFeatures realizes such first-class entities and at the same time is a pure Ruby language extension which can be used with any Ruby interpreter.

The first step in using rbFeatures is to define the features. This is done with a very concise notation: objects of class Class simply include the Feature module (see Figure 5).

```
1 class Weighted
2   is Feature
3 end
4
5 class Unweighted
6   is Feature
7 end
```

Figure 5: Defining Basic Features

The next step is the actual implementation of the product line. Code which is associated with a feature is expressed as *feature containment*. A containment consists of a *containment condition* and a *containment body*. The condition again is a set of named features and modifiers which expresses if the feature needs to be activated or deactivated. The body is a block which encompasses any scope: Whole blocks of code, single lines, and even parts of a source line. Conditions determine whether the body is executed or not.

Consider the Figure 6. The WEIGHTED feature impacts the class Edge at two specific points. First, in line 3, the attr_accessor (defines getter and setter methods for an class attribute) for weight. Second, in line 5, which parses the arguments to the new operator and sets the weight appropriately.

```
1  class Edge
2    attr_reader :source, :sink
3    Weighted.code { attr_accessor :weight }
4    def initialize(params)
5      Weighted.code { @weight = params.delete :weight }
6      params.delete :weight if params.include? :weight
7      @source = params.first[0]
8      @sink = params.first[1]
9    end
10 end
```

Figure 6: Feature Containments

After finishing defining all features, the product line is ready to be used. The programmer interacts with the module FeatureResolver to initialize the program. The FeatureResolver receives a copy of the source code, parses it and defines all features and other needed entities. Then, users simply activate or deactivate any feature. Each time when the activation status of a feature changes, the whole product line is evaluated again which leads to updated method and class definitions.

A special property of rbFeatures is its provision of domain-specific error messages. Whenever the user tries to call a method which is not defined because of a failing feature condition, the user is informed with a comprehensive message. For example, if he tries to call the MST algorithms, but did not activate the weighted features, the message is "*FeatureNotActivatedError: Feature Weighted is not activated*".

More details regarding overall workflow, implementation details and the product line composition can be found in [GuSu09].

5 Outlook

This short report gave a overview to the research in DSL engineering. We motivated our research with the two ongoing problems of the *tyranny of the dominant decomposition* and the *language problem*. The central hypothesis is that realizing both domains and application languages as internal domain specific languages based on one base languages allows the seamless analysis, specification, implementation and testing of complex applications. Afterwards, we presented the five steps of our research. We showed that the foundation and practical language engineering phases are finished. Immediate next steps are the

researching existing Ruby DSL from the Rails application to form a catalogue of DSL patterns and to refine the process with which languages and applications are developed. As a side project, we will also look into developing DSL with SAP BlueRuby, a Ruby implementation on top of the ABAP stack[1]. Finally, we will develop a medium sized web application and proof our processes.

References

[Apel07] Apel, S.: The role of features and aspects in software development. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2007.

[Arno+95] Arnold, B. R. T.; Deursen, A. van; Res, M.: An algebraic specification of a language for describing financial products. In ICSE-17 Workshop on Formal Methods Application in Software, 1995, pp 6-13.

[Bato+04] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering, no. 30, 2004, pp 355-371.

[CoMa98], Consel, C.; Marlet, R.: Architecturing software using a methodology for language development. Principles of Declarative Programming, Springer, Berlin et. al., 1998, pp. 482—506.

[Cook+07] Cook, S.; Jones, G.; Kent, S.; Wills, A.:Domain Specific Development with Visual Studio DSL Tools. Addison-Wesley Longman, Amsterdam, 2007.

[Copl00] Coplien, J. O.:Multi-paradigm design. Dissertation, Vrije Universiteit Brussel, 2000

[CzEi00] Czarnecki, K.; Eisenecker, U. W.: Generative programming: methods, tools, and applications. Addison-Wesley, Boston et. al., 2000.

[Gree+04] Greenfield, J.; Short, K.; Cook; S.; Kent, S.: Software Factories - Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley Publishing, Inc., Hoboken, 2004.

[Groo+95] Groote, J. F.; Vlijmen, van S. F. M.; Koorn, J. W. C.: The safety guaranteeing system at station hoorn-kersenboogerd. In Proceedings of the Tenth Annual Conference on Computer Assurance, 1995, pp 57-68.

[GuSu09] Günther, S.; Sunkle, S.: Feature-oriented programming with ruby. In Workshop on Feature-Oriented Software Development (FOSD), 2009 (to appear)

[Huda98] Hudak, P.: Modular domain specific languages and tools. In Proceedings Fifth International Conference on Software Reuse (ICSR), 1998, pp. 134-142.

[1] https://wiki.sdn.sap.com/wiki/display/Research/BlueRuby

[Kaes08] Kästner, C.; Apel, S.; Kuhlemann, M.: Granularity in software product lines. In Proceedings of the 30th International Conference on Software Engineering (ICSE), 208, pp. 311-320.

[KeTo08] Kelly, S.; Tolvanen, J. P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Hoboken, New Jersey, 2008.

[Kicz+97] Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. Proceedings European Conference on Object-Oriented Programming (ECOOP), 1997, pp. 220–242.

[Land66] Landin, J.: Then next 700 programming languages. Languages, Santa Barbara, 1966, pp. 11–26.

[Latr+07] Latry, F.; Mercadal, J., Consel, C.: Staging telephony service creation: A language approach. In Proceedings of ACM Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm), New York, 2007.

[LoBa01] Lopez-Herrejon, R. E.; Batory, D.: A standard problem for evaluating productline methodologies. Springer, Berlin, Heidelberg, 2001, pp. 10–24

[Mern+05] Mernik, M.; Heering, J.; Sloane, A.: When and how to develop domain-specific languages ??? 37:316–344, 2005.

[MeOs04] Mezini, M.; Ostermann, K.: Variability management with feature-oriented programming and aspects. ACM SIGSOFT Software Engineering Notes, no. 29, 2004, pp. 127–136.

[MuCl07] Munnelly, J.; Clarke, S.: Alph: a domain-specific language for crosscutting pervasive healthcare concerns. In Proceedings of the 2nd workshop on domain specific aspect languages (DSAL), Vancouver, 2007.

[Preh97] Prehofer, C.: Feature-oriented programming: A fresh look at objects. Springer, Berlin et. Al., Lecture Notes in Computer Science, vol. 1241, 1997, pp. 419-443.

[Spin01] Spinellis, D.: Notable design patterns for domain-specic languages. Journal of Systems and Software, no. 56, 2001, pp. 91–99.

[Sunk+08] Sunkle, S.; Rosenmüller, M.; Siegmund, N.; Rahman, S. S. ur; Saake, G.; Apel, S.: Features as first-class entities - toward a better representation of features. In Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, 2008, pp. 27–34.

[Tarr07] Tarr, P.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[Tarr+99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M. J.: N degrees of separation: Multi-dimensional separation of concerns. In Proceedings of the International Conference on Software Engineering, 1999, pp. 107–119.

[Thib+97] Thibault, S.; Marlet, C.; Consel, R.: A domain-specific language for video device drivers: from design to implementation. In Proceedings of the 1st USENIX Conference on Domain-Specific Languages, Santa Barbara, 1997, pp. 1997.

[Deur+00] Deursen, van A.; Klint, P.; Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, no. 35, 2000, pp. 26-36

[Viss08] Visser, E.:Webdsl: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, Generative and Transformational Techniques in Software Engineering (GTTSE), Lecture Notes in Computer Science. Springer, Springer, Berlin et. Al., vol 4143, 2008.

[With96] Withey, J.:Investment analysis of software assets for product lines. Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI96-TR-, no. 010, 1996.