Design Principles for Internal Domain-Specific Languages: A Pattern Catalog illustrated by Ruby

Sebastian Günther

School of Computer Science, University of Magdeburg

Thomas Cleenewerck

Software Language Engineering Lab, Vrije Universiteit Brussel

Abstract

Dynamic programming languages offer an infrastructure for the construction of internal domain-specific languages (DSL). DSLs inherit the facilities of their host language such as the availability of libraries, frameworks, tool support, and other DSLs. When developing an internal DSL, there are two challenges. First, to cope with the host language's syntactic and semantic restrictions. Careful and thoughtful extensions and modifications of the host language are crucial to overcome these restrictions without reverting to poor language design. Second, to support several design principles that are genuine for a DSL. Although there is an extensive body of knowledge about DSL design principles and desirable quality properties, it remains difficult to apply them, or to reason about whether a particular DSL exhibits specific desirable principles.

Our objective is to put the two perspectives together. We research the most important design principles of a DSL and show how different patterns can be used to support these principles. This allows us to produce a rich pattern language describing a principled approach for designing internal DSLs. The pattern language can be used to assess the design quality of a DSL and structure its implementation. In particular, we show a complex DSL example illustrating each principle and the corresponding patterns. While we stick to Ruby for the explanation and application of the patterns, we also name known uses in Python, Scala, and Smalltalk. Patterns are explained with their context, problem, forces, solution, and consequences. We also explain the patterns with the classical structure of intent, motivation, forces, implementation and their consequences. Finally we reflect upon the pattern utilization by discussing their strengths and weaknesses.

Keywords: Domain-Specific Languages, Design Principles

Email addresses: `sebastian.guenther@ovgu.de` (Sebastian Günther), `tcleenew@vub.ac.be` (Thomas Cleenewerck)

1. Introduction

Domain-specific languages are programming languages or executable specification languages that offer, through appropriate notations and abstractions, expressive power tailored for a specific problem domain or application area [47, 25]. At the implementation level, two different types of DSLs can be identified. Specially crafted external DSLs require their own parsers, interpreter or compilers, while internal DSLs are built on top of an existing programming language (called the host language), allowing them to reuse the host language's infrastructure including IDEs and compilers [33]. DSL are a common practice in software engineering as several examples show [2, 20, 45, 29, 35].

The wide spectrum of domains, internal DSLs, and external DSLs is threatening to deteriorate the original meaning of the term DSL. However, analyzing the relevant literature [33, 44, 27, 49, 11, 9] reveals that there are several design principles of DSLs, such as *abstraction* to make domain concepts explicit programming language constructs, *absorption* to integrate implicit and common domain knowledge in the DSL, or *compression* to yield a concise domain notation.

While these principles are general enough to provide guidelines for the design and implementation of a DSL, it is rather hard to explain whether an implemented DSL supports or embodies these principles. It is also difficult to plan the implementation upfront without a sophisticated language that allows expressing design decisions with respect to the principles. Hence the question: how to achieve this? We argue that the careful selection and application of patterns help here. Patterns support the DSL implementation with embodying these principles. By providing a deep understanding of the design principles together with a pattern language, we facilitate the planning and implementation of DSLs that exhibit the design principles.

This paper's contribution is a rich pattern language describing a principled approach for designing internal DSLs that can be used to assess the design quality of a DSL and structure its implementation. To the best of our knowledge, this particular combination of principles and patterns has not been elaborated in other literature yet. For feasibility of this first-hand analysis, the paper focuses upon selected principles and illustrates the application of patterns grouped into form (alternative syntactic forms for the expressions), modification (small changes to the language, like the provision of aliases), and support. The patterns are illustrated using the Ruby programming language and two internal DSL designed within this language. However, the patterns are available in other programming languages too, which we show for Python, Scala, and Smalltalk by pointing to known uses.

The paper is intended for novice as well as experienced DSL engineers. For experienced DSL engineers our work consolidates the design qualities of their DSLs enabling them to learn and improve their DSL designs. Novice DSL designers can use the principled pattern approach as a reference work to create internal DSLs.

We structure the paper as follows. Section 2 explains the used pattern structure and grouping. In Section 3, the design principles are explained. Sections 4 to 6 list the patterns with their intent, motivation, forces, structure, consequences, and additional known uses outside the herein presented examples. Section 7 revisits the design principles and, using a larger DSL examples, shows how the patterns are used in combination to design a DSL with respect to the principle. Section 8 discusses our experiences in using combinations of patterns and their influence on all principles. Section 9 explains the

related work and finally Section 10 summarizes this paper. We apply following formats: *keywords*, *design principles*, and PATTERNS.

2. Pattern Structure and Common Example

2.1. Pattern Structure

The pattern explanation has a structure that is close to the initial introduction of patterns [19] and recent publications [21, 12, 51]. The following structure is used to explain each pattern:

- *Context* – Explains the coarse situation in which the pattern is applied.
- *Problem* – A question expressing the central concern the pattern addresses.
- *Forces* – Short description in what circumstances this pattern should be used and the resulting effects, with an explanation of how the force is connected with a design principle.
- *Instructions* – Explanation what language mechanisms are used to provide a solution.
- *Example* – Based on the common examples, a detailed utilization of the pattern using the Ruby programming language is given. Source code examples and sometimes diagrams provide further illustration (see ►Figure 1 for used symbols).
- *Consequences* – Explains how this pattern's utilization impacts the DSL implementation or the DSL itself, and also explains possible tradeoffs and pattern combinations.
- *Known Uses* – List of applications that use this particular pattern, given for Ruby [17, 46], Python [31, 43], Scala [36, 48], and Smalltalk [40, 6, 7]. The analyzed DSLs are listed in the next sections.
- *Related Pattern* – Finally, a list of closely related patterns.

2.2. Analyzed DSLs

Following DSLs were analyzed for finding the known uses for the patterns.

- Ruby
  - *Rails* – Web-application framework with a strong Model-View-Controller architecture. Used in several open-source and commercial projects. In addition we analyzed the *Builder* library that is used by Rails to build configuration objects representing XML-like data structures (http://rubyonrails.com).
  - *ActiveRecord* – Rails standard database abstraction (http://ar.rubyonrails.com)

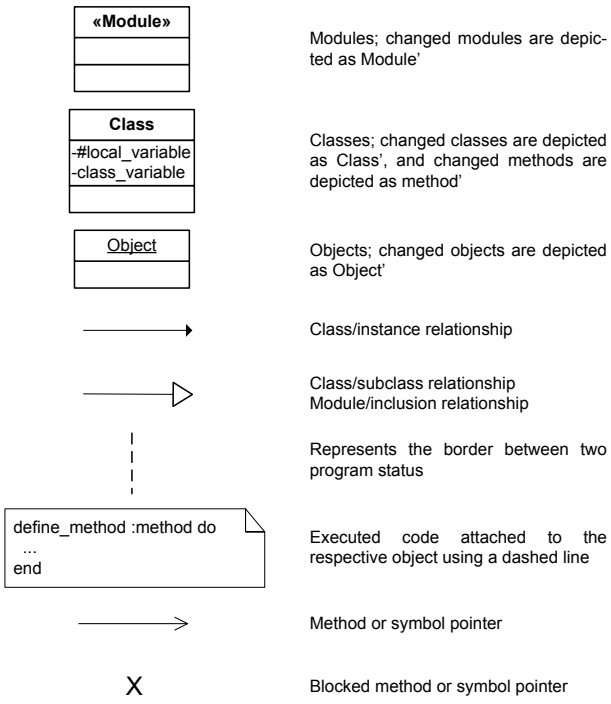

Figure 1: Symbols used to explain the pattern's structure.

– *RSpec* – Framework and DSL for behavior-driven development (http://rspec.info).
– *Sinatra* – Lightweight web framework that uses declarative expressions which look like a DSL for web applications (http://sinatrarb.com).

- Python
  - *Bottle* – Similar to Sinatra, a lightweight web framework that uses normal Python function declarations and annotations to specify how queries are responded in the application (http://bottle.paws.de/).
  - *Should DSL* – Supports simple behavior-driven testing by adding expressions like `should`, `be`, and `be_greater_then` as expressions to Python (http://github.com/hugobr/should-dsl).
  - *Cryptlang* – A DSL for defining cryptographic algorithms and random number generators [1].
  - *Constellation Launch Control DSL* – Provides a command like language for "programming test, checkout, and launch processing applications for flight and ground systems" [4].
- Scala
  - *Actors* – Actors is a build-in Scala DSL for asynchronous messaging and frequently deployed for concurrent programming. It uses concepts like a messaging box and special operators used to send and retrieve messages (http://www.scala-lang.org/api/current/scala/actors/package.html).
  - *Spec* – DSL for behavior driven development similar to the block-like nature of RSpec (http://code.google.com/p/specs/).
  - *Apache Camel* – Helps to express the configuration of enterprise systems that consist of java-compliant applications, allowing to define routes and APIs for the components (http://camel.apache.org/scala-dsl.html) Implementation of SQL as a Scala DSL [41].
- Smalltalk
  - Mondrian is an information visualization engine that lets the visualization be specified via a script. It is built in VisualWorks Smalltalk and Pharo (http://www.moosetechnology.org/tools/mondrian).
  - Glamour is an engine for scripting browsers (http://www.moosetechnology.org/tools/glamour).

2.3. Pattern Grouping

The patterns are grouped in three classes to distinguish their original concern. *Form* patterns allow to use alternative syntactic forms for the expressions, which can also change the basic paradigm expressions stem from (like the change from imperative expressions to declarative expressions using functional programming). *Modification* patterns change the language on a smaller scale like to provide aliases for existing entities. Opposed to the form pattern, modifications can also require semantic changes like to overwrite existing methods. Finally, the *support* pattern's role is to help the other patterns in achieving their respective goals.

2.4. Common Examples

2.4.1. Product Line Configuration Language

Software product lines address the challenge of structuring and systematically reusing software by providing a set of valuable production assets [15]. Such 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" [50]. Features describe modularized core functionality of a software [3].

Software product lines can be modeled as a feature tree [15] – it structures the features, their relationships, and the constraints they impose on each other (like to choose from a set of alternative features). A sample feature tree, showing the Graph Product Line (GPL) case study explained in [30], can be seen in ►Figure 2. This product line represents different types of graphs (weighted/unweighted and directed/undirected) and their corresponding property and search algorithms.

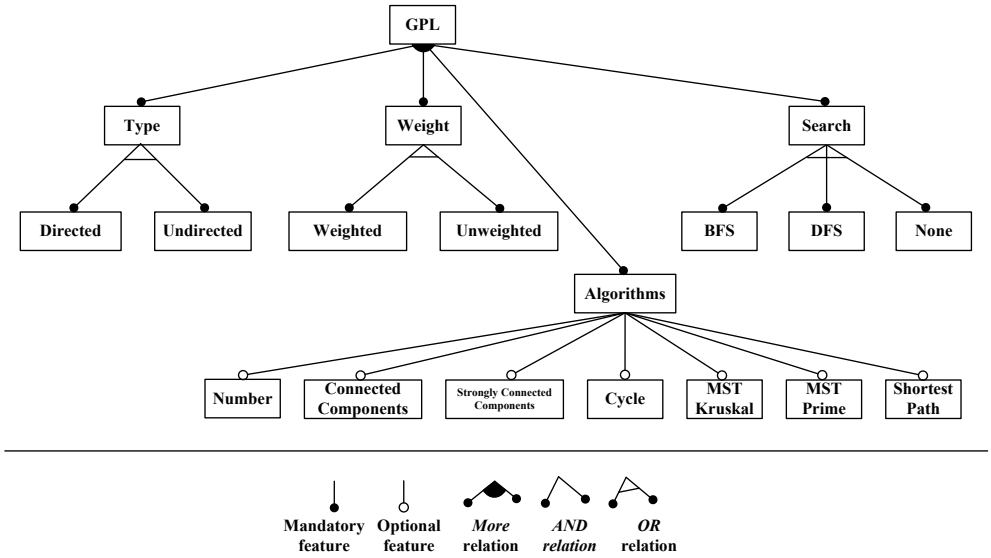

Figure 2: The Graph Product Line example from [30].

In order to model such feature trees, we designed the internal Ruby DSL called Software Product Line Configuration Language (SPLCL). The SPLCL uses a declarative textual notation to specify features with their name, position in the feature tree, subfeatures, and constraints. The DSL was developed as a research project by one of this paper's authors [22] and has since then continually evolved and served as vehicle for DSL integration research, amounting to four men months of work effort.

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

Figure 3: Example of the SPLCL DSL to configure the root feature of the GPL.

►Figure 3 shows an excerpt from this DSL how the root feature is configured. Here

is the meaning of this code:

- *Line 1* – Calling a constructor for a feature which receives configuration options in the following lines.
- *Line 2* – Set the name of the feature to GPL.
- *Line 3* – Express that the feature is at the root position.
- *Line 4* – Configure the subfeatures.
- *Line 5* – A constraint in order to activate this feature. It literally expresses that when the `GPL` feature is activated, the features `Type`, `Weight`, `Search`, and `Algorithms` have also to be activated.

2.4.2. Infrastructure Management DSLs

Infrastructure management takes care of the initialization, configuration, and maintenance of several servers forming the infrastructure of a stakeholder. Because of continuously increasing requirements with regard to flexibility, high automization support that limits the amount of manual involvement becomes crucial. We designed a set of three DSLs that support various tasks in this domain. The *Boot-DSL* identifies and installs machines, the *Software-Deployment Planning DSL* (SDP-DSL) expresses relationships between packages, and finally the *Configuration Management DSL* (CM-DSL) configures and installs packages. The currently supported operating systems are Linux-based. The supported hypervisors are Amazon EC2[1] and VMware ESX[2]. The DSLs are the collaborative effort of two bachelor student's theses and one of this paper's authors ongoing research project, amounting to more then two men years.

```
1  app_server = Machine "Application Server" do
2    type EC2
3    owner "sebastian.guenther@ovgu.de"
4    os :debian
5    hypervisor do
6      ami "ami-dcf615b5"
7      source "alestic/debian-5.0-lenny-base-2009..."
8      size :m1_small
9      securitygroup "default"
10     private_key "ec2-us-east"
11   end
12   hostname "admantium.com"
13   monitor :cpu, :ram
14   bootstrap!
15 end
16
17 deploy "Redmine" do
18   enroll "Redmine/Database", :on => app_server + aux_server do
19     #...
20   end
21 end
```

Figure 4: Example of the Boot-DSL and the SDP-DSL to configure a server and installs a package on it.

[1] http://aws.amazon.com/ec2/
[2] http://www.vmware.com/products/esx/

►Figure 4 shows an example for the Boot-DSL and the SDP-DSL, with the following meaning:

- *Line 1 – 15* – An example of the Boot-DSL that opens a constructor, sets the default name, and passes other configuration options in the form of a block.
  - *Line 2 – 4* – Configures the type, owner, and the operating system of this machine.
  - *Line 5 – 11* – A nested block that specifies the hypervisor-specific options like the amazon machine image (ami) id and its source, the size, the securitygroup (determines the available services and the open ports of the machine, such as SSH on port 22), and the private key used to access this machine via SSH.
  - *Line 12* – The hostname of the machine which is used inside the network.
  - *Line 13* – Expresses to monitor the CPU and RAM usage of this machine.
  - *Line 14* – Finally an expression to install basic software.
- *Line 17 – 21* – An example of the SDP-DSL to deploy the package Redmine (an open-source project management application, see http://www.redmine.org/).
  - *Line 18* – Redmine has several dependencies to satisfy, and in this case the database is configured to be enrolled on the application server as well as an auxiliary server for obtaining a master-slave relationship. Other packages would be implemented likewise.

3. Design Principles

General purpose languages were designed for expressiveness and scalability to solve a large number of computational problems. They are more concerned with providing efficiency and use language constructs which are independent of a particular application domain. Instead, DSLs absorb concepts of the problem domain directly into the solution domain, and language constructs are used to represent the domain. But this requires a thoughtful approach.

Dynamic programming languages, such as Ruby and Python, provide several options to implement DSLs. Through several syntactic alternatives, support for multiparadigm-development, and metaprogramming capabilities, these languages can be customized in terms of their syntax and semantics. However, it is challengingly to know the forms of these modifications and to use them coherently in the process of designing a DSL.

Studying works from the language design community, the software engineering community, and the domain engineering community, we see that a vast amount of principles is offered, but they are rather vague and non-constructive as the following examples show:

- The language design community focuses more on qualities. The important exception is multiparadigm programming by Coplien, where he defined abstraction as a major design principle and gave it a constructive definition in the context of communality and variations [11].
- In software engineering, it is widely acknowledged that “Inside every domain-specific framework, there is a language crying to get out” [18]. But how to construct a language out of packages, classes, and methods is far less obvious.

- The domain engineering community occasionally hints at a principle: classes are said to represent the vocabulary of the problem domain [8] and FODA suggests to apply aggregation/decomposition or generalization/specialization to the end of obtaining common assumptions [26].

In order to better guide the design of the languages behind DSLs, we analyzed the existing DSL literature [47, 5, 28] to discover the following defining characteristics: DSLs are (1) small, (2) concise programming languages or executable specification languages that offer, through (3) appropriate notations and (4) abstractions , (5) expressive power (6) tailored and (7) optimized towards a specific problem domain or application area. Each of those characteristics are embodied by the following design principles:

Generalization

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

In software engineering and especially domain engineering, generalization plays an important role. First, the domain is captured in its full scope consisting of multiple concepts and their relationships. They naturally form a hierarchy in which one concept is more general then other specific concepts. Finding such hierarchies helps the developer to understand the domain. The developer then chooses how to prune, reorganize, and combine the concepts in order to express generic and specific cases. Generalization keeps DSLs small as the reduction of the amount of concepts results in a small amount of needed constructs [11].

Compression

Provide a concise language that is sufficiently verbose for the domain experts.

The goal of compression is to reduce the amount of expressions or to simplify their appearance while the semantics are not changed. Compression (or compactness) makes DSLs concise programming languages that are still verbose enough for the domain experts. This reduces the code footprint of DSL programs [27, 49].

Representation

Provide notations and abstractions both required and suitable for the domain.

A DSLs representation is its syntax. The syntax consists of all language constructs, keywords, structures, and rules governing their combination. The rules determine the general layout of the language and, being an internal DSL, can also include hooks where other DSLs and host language expressions can be put. Internal DSLs cannot leave the syntactic frontiers of their host language, but they can chose to combine existing syntactic modifications to achieve a form that has little or no resemblance to the host language. Representation equips a DSL with the appropriate notations [47] by defining the basic syntax of the language and its grammar to support domain-specific abstractions and notations.

Abstraction

Form the DSL to contain common assumptions about the domain and deemphasize (technical) details.

DSLs abstract twofold: From the domain and from their host language. Abstracting from the domain is to include only those entities and operations in the language that are relevant. Abstractions from the host language are either base level abstractions or meta level abstractions. On the base level, the hosts basic operational model expressed by the existing language constructs is extended with novel expressions. At the meta-level, the existing constructs are tweaked, to modify or to extend their behavior. Both forms of abstractions are fundamental for DSL since they have to step away form the language in order to step towards the domain. Abstraction provides DSLs explicit focused expressive power [11].

Absorption

Implicitly absorb domain commonalities to facilitate DSL usage.

Characteristic properties of domains are commonalities for a broad set of expressions. Users are supported effectively if these commonalities are absorbed by the language meaning that certain assumptions and concerns are not required to be explicitly expressed. DSLs with high absorption support the user by capturing these assumptions implicitly, relieving him of the burden to repeat himself. This is in stark contrast to a traditional library, where functions are dependent on each other, require a manual setup of the context, and have a restricted call order. The best example are declarative DSLs that just describe a desired state, and the DSLs implementation uses all explicitly and implicitly given information to effectuate this desired state. Absorption provides DSLs implicit focused expressive power [44] by implicitly integrating domain commonalities in the DSL.

Standardization

Standardization tailors a DSL by restriction to able to assist users in how to write a program that will solve their problem.

Language constructs express language concerns. For example, object-oriented programming languages support declaration of classes, instantiation, provision of instance variables, and more. The focus on the essential concerns of a programming language and the formulation of as few language constructs as possible to express these concerns is called involution in programming languages [65]. DSLs aim for a high involution, meaning that the language will be standardized to a restricted set of operations and expressions based on problem domain. A significant part of their grammatical and semantic specification is therefore dedicated to assist users in how to write a program that will solve their problem. Standardization guides developers in using the tailored and thus restricted domain-specific set of language constructs [49, 32].

Optimization

Use algorithmic optimizations to improve the DSL's computational performance.

Producing performant code is an essential characteristic of programming languages. In the historic development of computer languages, the successful transition form hand-written assembly code to Fortran and its compiled machine code could only be made by showing that there are no performance decreases [39]. Likewise, an internal DSL should not be slower then its host language. Also from the domain side, the

provision of domain-specific optimization is important. Optimization improves the computational performance by using suitable models and domain-knowledge with an efficient implementation [9].

The relationships between patterns and principles will be explained in the next sections. For feasibility, we focus upon four principles that are more concerned with a DSLs syntax[3]: *representation*, *absorption*, *compression*, and *generalization*. Because the principles are heavily interwoven with other, we can not give a clear hierarchical decomposition of principles and their supporting pattern. Moreover, one pattern usually affects several other principles. Therefore, the paper continues with a focused explanation of the patterns grouped into the three categories *form*, *modification*, and *support*. Afterwards we will revisit the selected principles and explain how the different patterns support them. And finally, we combine both perspectives and explain our experience in using combinations of patterns and their influence on the principles.

[3]We do not explain *abstraction* because of its inherent semantic character which requires to fully explain the programming language's semantic details and metaprogramming mechanisms. We also skip *optimization* since this principle is very dependent on the domain. For example if graphs are represented in matrix form, various graph-specific algorithms can be used that are hard to generalize to other domains. Finally, *standardization* requires to create custom errors and check the semantics of executions, which is also not in the syntactic focus of this paper.

4. Form Patterns

Form patterns are providing the general layout of DSL expressions, they provide the framework in which expressions with a smaller scale are used. Patterns in this category also try to change the expression form the host languages basic paradigm to another one (like the change from imperative expressions to declarative expressions using functional programming).

4.1. Block Scope

Context

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

Problem

How to provide a lexically separate execution scope for expressions and how to stack hierarchical information?

Forces

- *Representation* – Group related expressions together and syntactically express the hierarchy of objects as indented blocks of code.
- *Absorption* – Absorbs the most relevant execution information, especially if it serves as a closure capturing the surrounding context and using them later to infer contextual information.
- *Compression* – Remove fully-qualified method calls inside a block, and let the execution context be determined by the object or method receiving the block.

Instructions

Ruby allows to group expressions inside a block using the **do** `...` `end` notation or curved brackets. The code contained inside this block has two distinguishing properties. First, it encloses its surrounding scope and can serve as a closure for specific execution states. Second, it can be executed in another context, using the variables from the surrounding lexical scope.

Block Scope is extensively used in other DSLs for these purposes. It can also be used as a Clean Room to execute all expressions in the safe scope of a specially crafted object [38], checking whether all method calls are valid or all values supplied correctly.

In Python, context managers can be used for the same purpose, although fully-qualified method calls still have to be used.

Example

Using the Boot-DSL to configure various virtual machine properties, the Block Scope is denoted by the **do** `...` `end` notation in Line 1 and Line 8. The properties are included as statements inside the block.

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

Figure 5: Block Scope: expression example that configures various properties of a virtual machine.

The block is attached to the `Machine` method. Its definition is shown below, starting with the `def` keyword. The method receives two parameters. The first is the machine's hostname, and the second is the block captured in the variable named `block`. Line 2 – 6 checks whether the machine already exists, and if not creates a new one. Then Line 7 evaluates the block in the machine's context.

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

Figure 6: Block Scope: implementation example. The block is captured as the variable name `block`, and then executed in the context of the created instance.

Consequences

- Possibility of code injection: If the block object is dynamically constructed with input from the users, malicious users could exploit detailed knowledge of the application to read its data or perform file system operations. However, Ruby has a good support for safe levels, as well as tainted and trusted objects [46], which reduces this threat's potential.
- Errors in the block can only be checked by executing the code, which is the task of the Clean Room.

Known Uses

- Ruby
  - Using Sinatra, Block Scope is applied with the declaration of request handlers, forming a declarative expression how a web application reacts to queried URIs (the request handler is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Line 752 – 758).

  - Using RSpec, Block Scope is applied for the declaration of example groups that begin with the `describe` keyword and group several test cases (implementation of the `describe` method is contained in rspec-1.3.0/lib/spec/dsl/main.rb, Line 24 – 29).

- Python
  - One of this paper's authors currently develops a SQL-DSL. Thereby, the *with-statement* is used – a syntactic extension of Python that allows a local variable to be passed to a set of indented expressions. The expression `with Query("sqlite")as query` starts a block where `query` is a local variable. Using this objects, methods representing SQL-statements are called. Because method calls inside the block still need to use fully-qualified names, its syntax is more verbose then the Ruby version altogether.

- Scala
  - In the Specs DSL, Block Scope is used to visualize the structure of complex test cases. Test cases, called system under specification in the Specs DSL, start with a literate String describing the system, followed by the `should` keyword and a function. Inside the function, examples are specified by using another string description and the `in` keyword. Interestingly, more keywords can be added to these terms, so that expressions like `"The␣web␣application"should have response status { ... }` can be built (the `should` method is defined in All Files specs-1.6.2.1/src/ main/scala/org/specs/specification/SpecificationSystems.scala, Line 45 – 46, and the `in` method in specs-1.6.2.1/src/main/ scala/org/specs/specification/ BaseSpecification.scala, Line 154.

- Smalltalk
  - Although closures are available in Smalltalk, each expression within a block needs a receiver. By combining message cascading with block closures like in `expression doWith: [ :result | result message1; message2; ... ]`) we can achieve similar expressions, but it is required to use a result argument as the start of a method cascade. The combination of message cascading and closures is not common. No examples have been found in the analyzed DSLs.

Related Patterns

- Keyword Arguments – Arguments passed to a method forms a grouping of expressions too, but they are usually executed in the scope of the method's body.

4.2. Method Chaining

Context

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

Problem

How to use method objects to mirror complex grammatical structures?

Forces

- *Representation* – Express the domain in a more natural-language like style by chaining method calls and their return values together.
- *Compression* – Use chained methods that transform an input value step-by-step, adopting a more functional-programming oriented style instead of more verbose imperative expressions.

Instructions

Normal method calls are written line by line and usually operate on passed variables. When the return values of one method are the input to another method, Method Chaining can be used. Because Ruby does not require fully-qualified method calls in the form of "receiver.method", the "`.`" can be left out, allowing to chain method calls together that look like language keywords. Multiple method chains enable free form language with a syntax that does not resemble Ruby.

Example

An expression that uses Method Chaining[4] is shown in the following listing. It is taken from the SPLCL to create a product variant (aka instance) for the GPL.

```
1 activate feature Weighted from productline GPL
```

Figure 7: Combining Method Chaining and Clean Method Calls for a complex yet easy to read expression.

The evaluation of this expression happens from right to left. Implementing Method Chaining demands to define various methods, often empty ones that are just used in the statement but have no semantic meaning. In considering the following implementation, the `productline` method receives the `GPL` object and sets a local variable (Line 2 – 4). Next, `from` is just an empty method (Line 7 – 11). Subsequently, `Weighted` is a method too, because of Ruby's lexical scoping. Therefore, the expression `feature Weighted`[5]is used to access the actual feature (Line 14 – 20). Finally, the `activate` method receives the feature and activates it (Line 23 – 25).

[4]And the later explained Clean Method Calls pattern.
[5]Actually an instance of the Custom Return Objects pattern.

```
1  # Setting the product line to the specified argument
2  def productline(arg)
3    @productline = arg if arg.is_a? ProductLine
4  end
5
6  # Providing empty methods that are just used inside the expression
7  def from(*args)
8  end
9
10 # Accessor for the 'Weighted' feature
11 def Weighted(*args)
12   :Weighted
13 end
14
15 def feature(arg)
16   Kernel.const_get(arg)
17 end
18
19 # Finally activating the feature
20 def activate(arg)
21   arg.activate
22 end
```

Figure 8: Method Chaining – implementation of all required methods in order to execute the method chain.

Consequences

- Requires to add several methods to the surrounding scope, possibly polluting the space with empty methods.
- Limited applicability to modify existing libraries because of the required extensive modifications.
- If used with an existing library, requires to change method internals.

Known Uses

- Ruby
  - In RSpec, users chain `should` and `should_not` expression with an assertion statement like `be_nil` or `have(3).items`, combining the statements in a clean and readable way (this works by adding the `should*` method to Ruby's built-in class `Object`, which is done in rspec-1.3.0/lib/spec/mocks/extensions/object.rb, Line 1 – 3).
- Python
  - The should-dsl is a unique extension because it uses Operator Expressions (explained later) together with a normal method call to add a new syntactic element to Python. Basically it allows the same kinds of expressions like RSpec, consisting of an object, a `should` keyword, and a test. Here is an example: `1 |should_not| be(2)`. Inside this expression, the "`|`" are actually method calls that receive the left value and the right value, which is then passed to the `should_not` method. This combination allows a nice-to-read Method Chaining (the "`|`" operators are defined in should-dsl/src/should_dsl.py, Line 25–38).

  - The Launch Constellation DSL also uses a form of *Method Chaining*: Methods with boolean return values are used to structure conditional blocks that determine how to react to different status of the monitored applications [4].

- Scala
  - The Specs DSL uses method chaining in an extreme form because expressions can mirror complete sentences. For example we create a car object with the name "Nissan Primera". To test whether the carname begins with the string "Niss" and not "Pors", this expression can be used: `nissan.carname must be matching("Nis*")and not be matching("Pors*")`. Similar matchers for objects other then strings are included in the Specs library or can be added for user-specific types (the `must` method is defined in specs-1.6.2.1/src/main/scala/org/specs/specification/Expectable.scala, Line 172, and the string matcher methods `be` and `matching` are defined in specs-1.6.2.1/src/main/scala/org/specs/matcher/StringMatchers.scala, Line 36 and 202)
- Smalltalk
  - Method chains come very natural in Smalltalk as one of the syntactic design goals of the language was the ability to write nice sentences. In mondrian, graphic shapes can be used as brushes using sentences such as `view shape rectangle size:`.
  - Using Glamour to create a browser for software versions, the expression `a list title: 'versions'`, where "`a`" is a widget constructor, is used to create a list with a title.

Related Patterns

- Keyword Arguments – Use key-value pairs instead of method calls to group related expressions.
- Block Scope – Group related expressions inside a block.

4.3. Keyword Arguments

Context

Methods are an important cornerstone in using DSL. They receive parameters which are expressing configuration of objects. However, there are two problems attached. First, always requiring a fixed order of passed arguments can provide problems if three or more arguments are passed. Second, just passing arguments hampers readers of DSL expression to relate the arguments with their intended meaning and relationship to the method.

Problem

How to provide a better readability of multiple arguments and how to provide flexibility in argument passing?

Forces

- *Representation* – Avoid complex API with methods receiving more then two parameters and the ambiguity of calling methods with correct arguments in correct order. Instead use keyword-value pairs as relationship declarations, to state mappings, or to declaratively express the desired system state. The chain of keywords and values can furthermore be used to obtain more natural-language like sentences.
- *Absorption* – Provide abundant optional arguments that are passed to other methods. Computations for good defaults for the optional keyword arguments absorb the additional information that can be deduced by the DSL itself.
- *Compression* – Compress several singular method calls to a set of key- value pairs passed to one method.
- *Generalization* – Easily switch to a specific or concrete semantics of a method call by using multiple dispatch on arguments.

Instructions

In Ruby, hashes provide a nice syntax for expressing relationships of objects: `key => value`. Hashes can be created as inline arguments to method calls. That allows to syntactically align the method call and its indexed arguments for better readability. Using these hashes inside the method calls to check for multiple arguments can be done either with the help of an iterator or by explicitly checking for certain keys.

Example

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

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

Figure 9: KEYWORD ARGUMENTS: the `enroll` statement uses keywords to configure deployment options.

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

```
def enroll(package, hash)
  @package = package
  @target = hash[:on]
  #...
end
```

Figure 10: KEYWORD ARGUMENTS: implementation of the `enroll` method which accesses the passed hash and set values accordingly.

Consequences

- Complex combinations of parsing arguments and error checking can make the method declaration cumbersome.
- More verbose method calls reduce *compression*.

Known Uses

- Ruby
  - Sinatra uses the `set` method for configuring options, it receives a single key-value pair and sets the configuration accordingly (the `set` method is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Line 622 – 636).
  - ActiveRecord uses the built-in `autoload` method that receives a module name as a symbol and a filename in which the module is implemented. Modules configured in this way are only imported when they are called in the program (the `autoload` method is defined in Ruby's core class `Module`, which is implemented in C for the MRI version of Ruby).
- Python
  - In Bottle, URL handlers are defined with normal Python functions and annotation. The annotations determine by which URLs the methods are executed. The annotation can contain additional keywords like in this example: `@route('/store/product/:id', method='POST')`. This defines a HTTP POST request to trigger the annotated method (the `@route` annotation method is defined as a decorator in bottle/lib/bottle.py, Line 562–566, and calls the method defined in Line 216–232).

  - The Launch Constellation DSL uses a variant of keyword passing. Instead of using hash structures directly, first strings are passed to the functions and then the values of the variables addressed by this string. Here is an example: `send_command (discrete ("VALVE1", "ON"))` [4].

- Scala
  - The analyzed Scala DSL do not use keyword arguments, but it is possible to implement variable length functions and pass any number of tuples inside, like in this contrived example: `configure_route("path"-> "/store/product/", "arg_arity"-> 2, "method"-> "POST")`. The arguments will form a Sequence which can be parsed conveniently with case matchers.

- Smalltalk
  - Keyword Arguments in Smalltalk can be realized by using method cascading, in which the receiver is stated at the beginning of an expression and where subsequent messages are send to the same receiver without having to repeat the receiver. This pattern is used in Mondrian on many occasions, pronounced examples are the sequencing of features of shapes e.g. they can be customized by a message cascade of `width:`, `height:` and `linearFillColor:` **`within:`**.
  - In Glamour, hierarchical browsers are created by a sequence of columns, each named after the message that has to be send to descend deeper into the hierarchy.

Related Patterns

- Block Scope – Key-value pairs can be expressed as method calls inside a block, but needs a method implementation for each.
- Method Chaining – Represent the key-value pairs as methods inside a chain, but requiring an implementation of each method too.

5. Modification Patterns

Modification patterns aim to change the host language expression for better compatibility with the domain, for example by renaming existing methods. Opposed to the form patterns, modifications can also require semantic changes like to overwrite existing methods.

5.1. Seamless Constructor

Context

In languages with an object-oriented core, the creation of new objects is typically expressed with keywords like **new**. In a DSL, using this keyword explicitly may not be intended, especially not when the keyword has another meaning. Many languages combine objects on-the-fly by using expression which uniquely expresses the type too. For example, entering any number in Ruby or Python expressions will create a number object with the same value.

Problem

How to create instances by just using the instantiating objects name and no explicit language-given keyword?

Forces

- *Representation* – Obtain a more fluently readable DSL program by omitting the need for explicit keywords to create instances.
- *Absorption* - Avoid the technicalities of creating new instances by absorbing the intention to create a new instance into the DSL.

Instructions

A simple solution is to use a custom method. The method receives parameters which configure several properties of an object, and then this object is returned. A more tricky solution is to use an expression which uses the very same symbol as an existing type. Ruby allows creating a global scope method with the same name as an existing type. The method simply returns a new object. Because of lexical scoping rules, the method can be used in places where method calls are expected (such as in Method Chaining), and thus we can create objects implicitly.

Example

The Boot-DSL uses a Seamless Constructor to instantiate new machines or to update the existing ones. ►Figure 11 shows that normally `Machine` points to the class, but afterwards, it points additionally to a method that returns a new or existing instance.

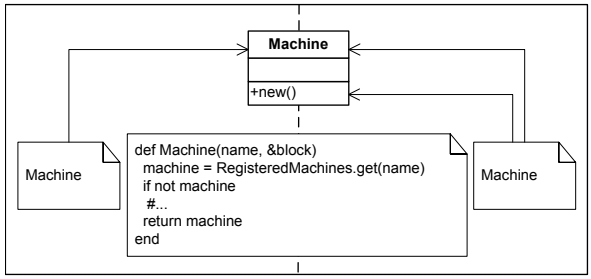

Figure 11: Seamless Constructor: providing a method with the same name as `Machine` that creates and returns a machine instance.

Consequences

- Lexical scoping determines whether the class or the method is meant when using the symbol.
- Method names written in capital letters violate Ruby conventions and could lead to confusion if not properly explained.

Known Uses

- Ruby
  - No example could be found in the other analyzed DSL, but they are used in the Boot-DSL and SPLCL as shown.
- Python
  - Per default, object instantiation is done by calling the object with attached parentheses. Python allows any variable to point to a method or a method call. This is used in the should-dsl: The method `should` actually calls `Should()` to create a new instance (the declaration of the `should` object is contained in should_dsl/src/should_dsl.py, Line 224).
- Scala
  - Scala supports Seamless Constructor with two options. Case classes are intended to serve as simple data containers and can be instantiated without the **new** keyword. The other option is to define a method that returns an instance, because Scala's type system distinguishes between the method call and the object instantiation even if the name is the same for both cases. We could not find this pattern in the analyzed DSLs.
- Smalltalk
  - Smalltalk allows defining class level methods that return an instance. This is used in Mondrian to implicitly create nodes that have to be drawn on the screen. Convenience methods exist to automatically create them given any list of elements.

Related Patterns

- Entity Alias – Similar to using a dedicated constructor, alias existing methods that create instances to hide the instance creation.

5.2. Entity Alias

Context

A programming language offers extensive functionality through its core-library and several external libraries. However, this functionality is usually expressed in terms which are suitable for the programmer. But these terms may have no or a misguiding meaning if used in a DSL. Implementing this functionality again is out of questioning. But still using appropriate names that fit the domain is crucial for the DSL.

Problem

How to change the names of entities such as modules, classes, and methods which stem from another library?

Forces

- *Representation* – Change the name of existing modules, classes, and methods to better suite the domain by creating aliases to customize the name of an external library or built-in class or method. When working with multiple DSLs, name clashes can be resolved of imported DSLs.
- *Compression* - If existing method or class names are frequently used, compress them with shorter aliases.
- *Generalization* - Provide alias names for already used entities when the internal object hierarchy changes.

Instructions

Ruby is very helpful with this kind of modification: the names of existing modules, classes, and methods can be modified arbitrarily. There are two subpatterns:

- Class/Module Alias – Alias a class or module simply by declaring a variable with the new name.
- Method Alias – Provide an alias for a method using an interpreter keyword or method.

Example

In the Boot-DSL example, the expression `type EC2` was used to denote the hypervisor for the configured machine. But this is actually Method Alias for the built-in `include` method. Executing this method includes a module and its methods to the particular machine instance, and it also configures the machine type.

The Method Alias can be implemented either by using the interpreter keyword `alias` or the method `alias_method`. Both methods first receive the name of the old method and then the name of the new method.

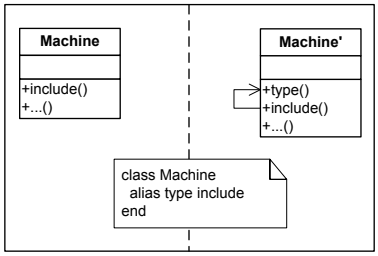

Figure 12: Entity Alias/Method Alias: providing the method name `type` as an alias for `include` inside `Machine`.

Class Alias are simple to implement. Any symbol in Ruby is just a global symbol registered in the symbol table. An assignment of the new name to the old name is sufficient. For example, to provide an Class Alias for `Machine`, the following declaration must be executed:

```
1 VirtualMachine = Machine
```

Figure 13: Entity Alias/ClassAlias: providing `VirtualMachine` as an alias for `Machine`.

Consequences

- Aliasing built-in methods can break compatibility with other libraries.
- Possible namespace conflict when executed globally.
- Sometimes, readability suffers if the used names are not obvious for other developers.

Known Uses

- Ruby
  - Rails has a central generator that is used to provide code stubs for models, controllers, and actions. The generators `initialize` method checks the passed parameter. If a controller is generated, the other passed parameters are calling the `action` method instead of **args**, thereby defining controller actions (the Alias Method for `initialize` is expressed in rails-2.3.5/lib/rails_generator/base.rb, Line 204).
  - Rspec allows to use the name `context` to describe an example group. This method is just an alias for `describe` (the Alias Method is expressed in rspec-1.3.0/lib/spec/dsl/main.rb, Line 30).
- Python
  - Python makes building aliases easy since each class or method can be selected from a running program (e.g.using the known method name or using reflection capabilities) and assigned to a new name. This is commonly called decorator in Python, and used for several methods in Bottle (bottle/lib/bottle, line 543–578).

- Scala
  - Similar to Python, there is no direct language support for aliasing methods, but manual re-declaration of methods is straightforward to implement. The Specs DSL implements a mechanism that works like an alias. Normally the `should` keyword begins a test case. With the execution of `def configure = addToSusVerb("configure")` the call can be extended to `should configure`, where configure has the same behavior as should. However, `configure` can not be used instead of `should`.
- Smalltalk
  - For implementing Method Alias, new methods can be attached to foreign classes (classes belonging to other packages) using extension packages. One example in Mondrian is the `Point>>translatedBy:` method, which has been defined for compatibility reasons as it is often used in layouts.
  - Class Aliases can be easily added by simply creating another binding in the globals pool of Smalltalk using the expression `Smalltalk at:#MyClass put:`**Class**. Another method is to create a proxy class that intercepts every message and redirects it to another object by overriding the method `doesNotUnderstand: aMessage`. However, we could not find such mechanisms in the analyzed DSLs.

Related Patterns

- Seamless Constructor – Provides an alias for the **new** method by aliasing a class name as a method.

5.3. Operator Expressions

Context

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 performing the set operations difference and joining with the symbols `-` and `&`. These symbols are perfectly suited to express the semantics of similar domain operations.

Problem

How to define symbols like `+`, `*`, `&` for DSLs?

Instructions

In Ruby and Python, most symbols are just method calls. Defining these methods in the classes that are used to call them will greatly enhance the domain specific look of DSLs.

In Ruby, following operators can be redefined [17]:

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

Table 1: Redefinable operators in Ruby (from [17]).

Forces

- *Representation* - Add domain-specific operations by using appropriate symbols instead of text for operations or relationship declarations.
- *Compression* – Provide shorter alternatives for frequently used method keywords.
- *Generalization* – Common operators can be used for multiple dispatch of arguments, providing different behavior upon the type of their arguments.

Structure and Implementation

The SDP-DSL expresses how applications are installed on machines. Applications can be enrolled on multiple machines, and to express this, the symbol "`+`" is used. Here is the example expression.

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

Figure 14: Operator Expressions: Using the "`+`" symbol to combine two or more machines on which applications are installed.

Since this operator is just a method, it needs to be defined in the `Machine` class, and additionally in the built-in `Array` class in order to chain the returned arrays together. This modification is shown in the following figure.

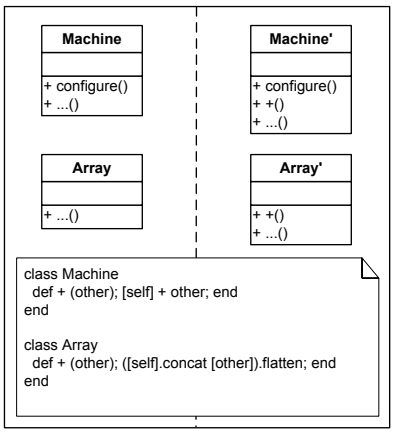

Figure 15: Operator Expressions: adding the operator "`+`" as a method to the class `Machine` and `Array`.

Consequences

- Operator expressions are just normal method calls, so they need to take care of different argument types and more.
- Some operators are required to be defined on other classes of the application or core library classes too.

Known Uses

- Ruby
  - ActiveRecord uses "`<<`" to add records to associations (defined in activerecord-2.3.5/lib/active_record/associations/association_collection.rb, Line 111–125).
- Python
  - Cryptolang defines 21 operations to express cryptographic algorithms with a very concise syntax. For example, inside the declaration of the blowfish algorithm, the expression `y = y ^ (bs[2] $ S[2])` combines logical-or expressions ("`^`") and a substitution ("$") by swapping array values [1].

- Scala
  - The Actor DSL uses several methods that are used to fill and query the mailbox object. The method "`!`" sends a new message, and "`!?`" sends a message and waits for some milliseconds for a synchronous answers. Messages on the stack can be queried with the "`?`" method (these methods are defined in scala-lang-2.8.0/src/actors/scala/actors/Channel.scala between Lines 45–108)
- Smalltalk
  - Operators in Smalltalk are binary or unary methods. In Mondrian, "`->`" is used to construct edges and "`@`" for points.
  - Glamour uses "`->`" to select a particular value from a part of the browser. For example, in a browser for software configurations, the selected version is obtained by the expression `#versionsAndBaselines -> #selection`.

Related Patterns

- Alias Entity/Alias Method – Alias an existing method to respond to a operator.

6. Support Patterns

Support patterns contribute to the efforts of the other patterns by changing expressions in such a way that they look more separate from the host language.

6.1. Clean Method Calls

Context

In a DSL, it is important to change the syntax for a more natural representation of the domain. Programming languages often require explicit delimiters such as brackets. Except structuring more complex expressions, these symbols usually do not add to a DSL's readability.

Problem

How to eliminate unneeded parentheses in cases where passing arguments is unambiguous?

Forces

- *Representation* - Provide clear expressions that do not resemble their host language but look like keywords of a domain-specific language.
- *Compression* – Remove parentheses, especially for nested method calls, to compress expressions.

Instructions

Ruby allows to drop parentheses in all cases where no lexical ambiguity is present. For example, using any symbol will lead the compiler to look for a constant, but if it is a method (especially with the *Seamless Constructor*), then parentheses must be used. In most cases however, method calls can be dropped from Ruby.

This pattern is not available in Python because parentheses are always required in this language.

Example

Declaring the machines in the Boot DSL uses several methods that set according values. Two expressions are contrasted here. The first one, Line 2 – 6, uses parentheses and the curved brackets for showing blocks. The second form, Line 8 –12, leaves out the parentheses and it uses the **do** `...` `end` block notation.

```
# using parentheses
Machine("Application Server") {
  hypervisor(:ec2) {
    ami("ami-dcf615b5")
  }
}
# omitting parentheses
Machine "Application Server" do
  hypervisor :ec2 do
    ami "ami-dcf615b5"
   end
end
```

Figure 16: Clean Method Calls: removing the parentheses around method calls and using the alternate block syntax simplifies expressions.

Consequences

- Sometimes, explicit knowledge about host language keywords and DSL keywords are more appropriate to understand the DSL completely.
- Lexical scoping determines the meaning of symbols, so using key-value pairs and blocks with curved parentheses can introduce syntactic errors

Known Uses

- Ruby
  - In Sinatra, the above mentioned request handlers are typically declared using Clean Method Calls (the request handler is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Line 752 – 758).
  - In RSpec, the example groups are also without using parentheses, which leads to a declarative specification-like language of application behavior (the `describe` method is defined in rspec-1.3.0/lib/spec/dsl/main.rb, Line 24 – 29).
- Python
  - If a method is called without using parentheses, the Python interpreter treats this as an access to the method object and returns it. Parentheses must be attached in order to call a method – with some rare case implemented by the should-dsl. Because built-in operators are defined as methods and cleverly chained, an expression like `Collection.item_count()|should| be(22).` can be used. Here, the `should` is actually a method call too, but its parentheses have been skipped (the `should` method is defined in should-dsl/lib/should_dsl.py, Line 224).
- Scala
  - Scala allows to remove parentheses in some cases. If a method receives no arguments at all, then it can be called without arguments. Especially the Specs DSL uses this capability to write specifications like `should not match("String")`

  - The Apache Camel DSL uses Clean Method Calls at several occasions, like in the expression `"app1" throttle (3 per 2 seconds)to ("app2")`[6] which expresses the intent to only push 3 messages per 2 seconds from the endpoint app1 to app2. Also methods that receive only one argument can be used like this.

- Smalltalk
  - This pattern is not supported. Smalltalk parsers are quit rigorous and do not allow for much syntactic freedom. As a result there are parentheses all over the place.

Related Patterns

None.

[6] Taken from the DSL examples at http://camel.apache.org/scala-dsl-eip.html.

6.2. Custom Return Objects

Context

In a DSL, some methods may require multiple structured objects, such as an array, to be returned to the caller. However, accessing these objects from the outside directly binds the caller to the provided data structures. This is not only difficult to refactor, but also explicitly accessing array values result in an undesired syntax which is to tied to language internals.

Problem

How to express value access more verbosely for multiple structured objects?

Forces

- *Representation* - Utilize keyword access to structured data instead of relying on array positions, providing better readability.
- *Generalization* – Invoke specific semantics in the custom return objects, without changing the caller or the immediate expression environment, thereby providing generalization.

Instructions

Instead of returning a flat list of values, pack the values inside an object and provide accessor methods to them. Depending on the language, objects can be defined on the fly or even proxies for value-accessing method calls can be provided.

Example

Back in the Clean Method Calls pattern, a very verbose expression to activate a single feature was used: `activate feature Weighted from product line GPL`. The expression `feature Weighted` is a custom return object because it returns a specific object for the `activate` method. But as explained, this syntax needs two method calls. Alternatively, the following expression and implementation can be used.

```
1 def feature(name)
2   case name
3   when Weighted
4     #... modifiy the object
5     return modified_weighted
6   end
7 end
```

Figure 17: Custom Return Objects: the `feature` method receives an argument which is checked, customized, and finally returned.

The result of the expression `Feature.Weighted` is returned to the activate method which works as before.

In addition to returning concrete objects, anonymous `Struct` objects are available too. They are simple objects with custom values, define in a concise syntax. For example, if a class should be returned with the methods `to` and `from`, the following expression is sufficient.

```
1 Struct.new(:to, :from).new("Return value of 'to'", "Return value of 'from'")
```

Figure 18: Custom Return Objects – definition of an anonymous `Struct` class with two data fields.

Consequences

- Named classes pollute the global symbol table – use anonymous `Struct` objects instead.
- Border between method calls and objects diminishes, which could make it hard to read for programmers.

Known Uses

- Ruby
  - Using ActiveRecord queries, like `clients.find_by_name("Sebastian")`, returns single values or arrays of records, which are typically included in other statements too (the definition of the `find_by*` methods happens dynamically in activerecord-2.3.5/lib/active_record/base.rb, Line 1839 – 1961).
- Python
  - With variable length arguments, an appropriate class, and some metaprogramming for defining accessors to the passed data, objects serving as Custom Return Objects can be implemented. However, such approach could not be found in the analyzed DSLs.
- Scala
  - The Apache Camel DSL expression `"app1" throttle (3 per 2 seconds) to ("app2")` uses several Custom Return Objects. The expression `3 per 2 seconds` will (a) create a `RichtInt` type with the value 3, (b) on which the method `per 2` is called to return a `Period` objects, and (c) the `RichInt` and the `Period` objects instantiate a `Frequency` object on which the `seconds` method is called to return the `Period`'s seconds (the `RichInt` type is defined in apache-camel/components/camel-scala/src/main/scala/org/apache/camel/scala/RichInt.scala, and the other types and their methods in Frequency.scala and Period.scala at the same place).
- Smalltalk
  - Custom Return Object in Smalltalk is implemented as prototypical objects that receive a dictionary containing key-value pairs of selectors and closures. The selectors denote the method name and the blocks denote the method body. Upon intercepting a message to that object one can consult the dictionary, look up the matching closure and execute it. This pattern is used for passing Mondrian nodes along in a sequence of messages where each message can change the node resulting from the previous message. Attributes can be created and added on the fly as the nodes have an arbitrary list of attributes.

Related Patterns

- Method Chaining – An extension is to store several values along the chain and then return an object containing all stored values or being dependent on them (like a query).

7. Principles and their Supporting Patterns

This section details how the selected principles can be supported with the patterns. For each principle, the patterns are listed and a larger DSL example is shown to illustrate how the patterns can be applied in combination to design a DSL with respect to the principle.

7.1. Representation

Provide notations and abstractions both required and suitable for the domain.

Supporting Patterns

- Block Scope – Group related expressions together and syntactically express the hierarchy of objects be indented blocks of code.
- Method Chaining – Use long sentences to verbosely express the domain by chaining method calls and their return values together.
- Keyword Arguments – Use keyword-value pairs as relationship declarations, to state mappings, or to declaratively express desired system state.
- Entity Alias – Change the name of existing modules, classes, and methods to suite the domain.
- Seamless Constructor – Omit explicit keywords to create instances.
- Operator Expressions – Use symbols instead of texts for operations or relationship declaration.
- Clean Method Calls – Use method names that look like keywords of the language.
- Custom Return Objects – Use the returned object in another Method Chain or start a new Block Scope.

Example

The infrastructure management DSLs are used in ►Listing 19 to configure a virtual machine and to deploy an application. Using this example we demonstrate that the syntax of the DSL can be carefully designed by the above mentioned patterns to closely match a given domain-specific notation.

- *Line 1 – 15*: Using the Block Scope and Clean Method Calls patterns, the configuration options of a machine can be grouped and listed using keywords similar to domain-specific notation for such configurations.
- *Line 1*: A machine with the name "Application Server" is created. Using Seamless Constructor, this can be written as a declarative domain notation as there is no need to explicitly call **new** to create a new object.
- *Line 2*: The newly created machine is a `EC2` model. By using the Alias Entity pattern we can adopt the domain concept `type` into Ruby as alias for the built-in method `include`. As such, one expression is used both to include the module and to express which type the machine has.
- *Line 3–4*: The machine's owner and operating system are configured. Keyword-like methods are used to declaratively express properties.

```
 1 app_server = Machine "Application Server" do                Seamless Constructor
 2   type EC2                                                                Aliasing
 3   owner "sebastian.guenther@ovgu.de"
 4   os :debian
 5   hypervisor do                                                      Block Scope
 6     ami "ami-dcf615b5"
 7     source "alestic/debian-5.0-lenny-base-2009..."
 8     size :m1_small
 9     securitygroup "default"
10    private_key "ec2-us-east"
11   end
12   hostname "admantium.com"
13   monitor :cpu, :ram                                      Clean Method Calls
14   bootstrap!
15 end
16
17 deploy "Redmine" do
18   enroll "Redmine/Database",
19          :on => app_server & aux_server do    Keyword Arguments, Operator Expr.
19   #...
20 end
```

Figure 19: Representation: Example of the Boot-DSL (Line 1 – 15) to identify and configure a virtual machine, and the CM-DSL (Line 17 – 20) to deploy an application. Several patterns are used to form the DSLs.

- *Line 5 – 11*: Another Block Scope is used to configure the machine's hypervisor. The contained expressions for the hypervisor are logically grouped in accordance with the valid operation domain of the EC2 hypervisor.
- *Line 12 – 14*: More machine options are configured: The hostname, the monitored resources, and that this machine is bootstrapped (installs some basic software packages). Line 13 exemplifies that a cleaner and more domain-friendly syntax can be achieved by using Clean Method Calls to remove the parentheses from method calls (which has been used in the preceding lines as well).
- *Line 17 – 20*: Illustrates the SDP-DSL, expressing the intent to deploy the application Redmine[7] and its dependencies. Similar to Boot-DSL, the Block Scope and Clean Method Calls are the DSL's building blocks. By using indented blocks, the DSL expresses the natural hierarchy of the domain objects directly with language expressions.
- *Line 19*: The statement uses Keyword Arguments to better match the domain expressions by more verbosely stating that the Redmine/Database package is enrolled `on` the application server `and` an auxiliary server. Note the symbol "`&`" is actually a method call and an example of the Operator Expressions pattern.

[7] http://www.redmine.org/

7.2. Absorption

Implicitly absorb domain commonalities to facilitate DSL usage.

Pattern Support

Absorption depends on the application to have a rich amount of configured properties. The properties are then implicitly contained in language expressions, and the following patterns can be used:

- Block Scope – The scope's context absorbs the most relevant execution information, especially if it serves as a closure capturing the surrounding context and using them later to infer contextual information.
- Keyword Arguments – Absorb additional information as optional keyword arguments.
- Seamless Constructor – Absorb the intention to create a new instance.

Examples

In order to explain this design principle, the infrastructure management DSLs are used again. ►Figure 20 shows how the application server's hypervisor is configured.

```
 1 Machine "Application Server" do                          Seamless Constructor
 2   type EC2
 3   hypervisor do                                                Block Scope
 4     ami "ami-dcf615b5"
 5     source 'alestic/debian-5.0-lenny-base-2009...'
 6     size :m1_large, :limit => :costs                   Keyword Arguments
 7     securitygroup "default"
 8     private_key "ec2-us-east"
 9   end
10 end
```

Figure 20: Absorption: Boot-DSL and patterns that provide the absorption of context.

- *Line 1*: Expresses the customization of the application server. Whether a new machine is created or whether an existing machine is reconfigured is absorbed with the Seamless Constructor.
- *Line 3 – 9*: All expressions within the hypervisor block are executed against the specific hypervisor EC2. Using the Block Scope pattern, this dependency is absorbed into the inner `hypervisor` block.
- *Line 6*: Expresses to use a large size for the machine, and additionally that the total operating costs as a limit for the (re)sizing. If the costs are calculated as being to high, the small size will be used. The expression `:limit => :costs` is optional absorbed into the DSL by Keyword Arguments.

7.3. Compression

Provide a concise language that is sufficiently verbose for the domain experts.

Pattern Support

- Block Scope – Remove fully-qualified method calls and provide a named context for code execution.
- Method Chaining – Compress several method calls into a chain that operate on each other's return values.
- Keyword Arguments – Compress several singular method calls to a set of key-value pairs passed to one method.
- Entity Alias – Compress existing method or class names.
- Operator Expressions – Provide shorter alternatives for method keywords.
- Clean Method Calls – Compress expressions by removing parentheses, especially for nested method calls.

Example

In the following example, the GPL example is used to show three different variant how the root feature can be configured.

```
 1 gpl = Feature.new("GPL")
 2 gpl.tree_position(:root)
 3 gpl.set_subfeatures(:Type, :Weight, :Search, :Algorithms)
 4 gpl.add_requirments(self.name, "all :Type, :Weight, :Search, :Algorithms")
 5
 6 gpl = Feature.configure do                                          Block Scope
 7   name :GPL
 8   root
 9   subfeatures :Type, :Weight, :Search, :Algorithms
10   requires :GPL => "all :Type, :Weight, :Search, :Algorithms"     Keyword Args.
11 end
12
13 configure_feature :GPL, as_root do                              Clean Method Calls
14   subfeatures :Type, :Weight, :Search, :Algorithms
15   requires self => all(subfeatures)                              Method Chaining
16 end
```

Figure 21: Compression: Showing three different version for modeling features. The tree examples vary in the amount of compression that has been obtained. The first and last example depicts two extremes: the first one uses common object-oriented expressions and requires 181 characters, the last one uses 125 characters. A middle ground is found in the second example where compactness and readability are more balanced.

- *Line 1 – 4*: Normal object-oriented expressions, including method calls with parentheses. These expressions require 181 characters.
- *Line 6 – 11*: The current form of our SPLC, with a total amount of 162 characters. The Block Scope pattern is used to eliminate the fully qualified method calls as opposed to the first variant. Also, Clean Method Calls eliminate the parentheses. Finally Line 10 uses the Keyword Arguments pattern to provide a verbose but readable form of constraints.

- *Line 13 – 16*: A variant with high *compression* using only 125 characters. The name and position properties are specified as arguments to a method using Clean Method Calls (Line 13). This argument notation is similar to Keyword Arguments, but without using the keywords and thus allowing a very compressed form. In the Block Scope body, the subfeatures are configured as before, but this time combining Keyword Arguments with Method Chaining in Line 16. The constraint rule `all(subfeatures)` uses internal information about the feature to greatly compresses the expression.

It is difficult to argue what amount of *compression* is sufficient or useful for a particular DSL. For example, the first and second alternatives require similar amount of text, but the second version does not use symbols or method calls that are "typical" for programming languages. Although the third version is the most compressed form, the meaning of `self` and the method call `all(subfeatures)` implies knowledge about the host language – people not familiar with Ruby might understand the second alternative better.

7.4. Generalization

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

Pattern Support

Generalization is supported by providing entities and methods that serve as anchors for later specialization refinements. Therefore the following patterns are used:

- Keyword Arguments – Changing semantics by using multiple dispatch on arguments.
- Entity Alias – Provide alias names for used entities when the hierarchy changes.
- Operator Expressions – Common operators can be used for multiple dispatch of arguments, providing different behavior upon the type of their arguments.
- Custom Return Objects – The returned object is invisible to the caller. Semantic changes are confined to the objects without having to change the surrounding expressions.

Example

In the following ►Figure 22, example expressions from the SPLCL are shown.

- *Line 1 – 2*: An alternative form of configuring a product line that uses Keyword Arguments. The used keywords have the same meaning as the method expressions explained before. The expression has a high *generalization* because it focuses on the essential properties of software product lines, yet it is simple to extend with specialized behavior using new keywords.
- *Line 5 – 9*: This expression creates an actual variant of a product line.
  - *Line 6*: Configures the name of the particular product variant that is registered with the product lines.

– *Line 7*: A complex expression combining Method Chaining and Custom Return Objects. It expresses that this variant should use the product line `GPL`. The expression `feature(Weighted)` is return a custom object that selects the `Weighted` feature from the global namespace. This behavior can be changed transparently by the implementation without changing the language's syntax, thus facilitating *generalization*.

```
1 GPL.configure :root => true,                                    Keyword Arguments
2               :subfeatures => [:Type, :Weight, :Search, :Algorithms],
3               #...
4
5 variant.create do
6   name "SimpleVariant"
7   activate feature(Weighted) from productline GPL    Method Chain., Custom Ret.
8   #...
9 end
```

Figure 22: Generalization: using the SPLCL to configure a feature and then to create a variant. Patterns related to modifying and implementing methods are used to generalize expressions of the DSL.

8. Discussion of Pattern Usage

This section details our experience using the patterns and how they collaboratively support the principles.

8.1. Form Patterns

The form patterns are Block Scope, Method Chaining, Keyword Arguments.

Block Scope is a very dominant pattern. It's hard to find a Ruby DSL that does not use this pattern. The impact on readability achieved with Block Scope is fundamental: the **do** `...` `end` notation reads very natural and fits into several domains. It allows both to syntactically group related expressions and to express the natural hierarchy of objects. And to give the full potential of this pattern: even its semantic capabilities enable many modifications, including deferred evaluation, composition, and execution at any scope of the program. Block Scope can be used for the *representation*, *absorption*, and *compression* principle, but is of limited use for *generalization*.

Method Chaining and Keyword Arguments are both an alternative and an addition to the Block Scope. These patterns are not concerned with the syntactical grouping of related statements, but they express more complex statements. Since method calls can be used without using the fully qualified name and – using Clean Method Calls – can also drop all parentheses, chains of methods that look like language keywords, can be used. This enables more self explanatory statements that can be used for a clear representation of the domain.

The close connection between the form patterns are the following: A block allows to form expressions whose implementation is not available at the place they are specified,

but at the place they are executed. Thus, using Method Chaining and Keyword Arguments with Clean Method Calls in the context of Block Scope allows very free from languages. The overall result are verbose expressions used for the *representation* and also *absorptions*. Depending on the desired verboseness, *compression* can be limited by this combination. *Generalization* is strongly supported since multiple argument dispatch is possible with the Keyword Arguments and the Method Chaining.

8.2. Modification Patterns

The modifications patterns are Seamless Constructor,Entity Alias, and Operator Expressions.

The Seamless Constructor removes the need to use the **new** operator in order to create instances. The solution is to define a method with the same name as the class, and to use the class inside the method statement. The method can receive multiple arguments via Keyword Arguments that are used to customize the implicitly created instance. *Representation*, *absorption*, and *generalization* are supported.

The Alias Entity pattern can be used in a very similar fashion. Instead of eliminating a specific method call, a viable alternative can be provided. Alias Entity can be used to customize the whole Ruby programming language, since all classes and methods, even the built-in ones such as `Array` and `String`, can be modified to suit the domain. All design principles are supported, and if the modification of built-in methods is considered, *generalization* is especially well supported.

The final pattern here is Operator Expressions. Ruby is a language with a very minimal set of interpreter expressions. The creation of modules, classes, and methods are covered by this, and assignments. But beyond that, a rich set of operators can be defined since they are just method calls on objects. Domains that require such symbolic expressions benefit from the improved *representation* and *compression*.

8.3. Support Patterns

Support patterns are Clean Method Calls and Custom Return Objects.

Clean Method Calls is an extensively used support pattern. Although its intent is simple and its application trivial, it has a strong impact on *representation* and *compression*. Imagine to use Method Chaining while keeping parentheses: only because Ruby allows to eliminate most parentheses, expressions that look like using language keywords can be build. Clean Method Calls are primary supporting *representation*, and *compression* to the degree of removing parentheses.

Finally, Custom Return Objects is a support pattern for Block Scope, Method Chaining, and Keyword Arguments. The primary use case is to simply return an object and access its data fields. Additionally, it can be used inside a method chain too, enabling multiple dispatch arguments. Or it receives a block and can start a Block Scope. It is also an alternative for Method Chaining since the returned objects can also define arbitrary data and methods specifically in its scope without polluting the global namespace. Therefore, Custom Return Objects supports the *absorption*, *compression*, and *generalization* (its easy to customize the objects behavior) principle. Its effects on *representation* are usually only achieved in combination with other form patterns.

8.4. Summary

The explained patterns are clearly fitting into a narrower scope. While in isolation they provide isolated advantages for some design principles, their thoughtful combination supports several design principles. The form patterns provide the building blocks of a DSL. They characterize the overall *representation* of the domain as either (i) blocks of hierarchically structured expressions, (ii) grouped expression, or (iii) verbose and complex expressions. Thereby the form patterns use the support patterns to provide method calls without using parentheses or to use objects as an anchor for further block or complex expressions. The modification patterns support adaptation and extension of the available constructs of the domain, which greatly enhances the available forms.

In the pattern's description and the discussion, two facts emerged. First, a pattern supports multiple design principles. It is seldom feasible to use a pattern in isolation to support a single principle. More so, the principles itself are connected and strengthen each other. Second, the patterns have close relationships to each other and are typically used in combinations. For example to use Method Chaining without Clean Method Calls would severely impair readability of the expressions. This interleaving means design considerations commonly lead to a closely coupled set of patterns.

9. Related Work

General work on Domain-Specific Languages and their development starts with [42, 33]. These early works are proposing a complete design process including patterns. The patterns however are very coarse grained, like using Unix pipes and filers to chain command or the usage of an interpreter. Our work has more detailed focus on dynamic languages and syntactical language modifications according to design principles.

For graphical modeling languages, criteria how to select different visualization forms for models are presented [13, 34]. However, the close focus on graphical notation does not allow a generalization for usage in designing textual languages. There are some empirical studies regarding what concepts to include or not to include in a language [37, 24], but these approaches lack an overall conceptual framework while our pattern language offers a principled design method.

In related work in the context of DSLs and dynamic programming languages, few authors try to structure the techniques they use for the host language's modification. They mix language characteristics with concrete mechanisms and design decisions to explain their approach. For example, [14] explains a DSL for surveys using several Ruby metaprogramming mechanisms. [16] explains a Groovy-based DSL for aspect-oriented programming, using an internal interpreter object. For Python, DSLs to secure and restrict the access to resources [10] or for cryptography are available [1]. Also, most of these works are focused on semantic extensions of the used host language, considerations of the DSL's syntax are only seldom discussed, like in [1] that uses operator expression for the DSL. Exceptional is a book about the Scala programming language [48], where several implementation techniques are presented, including different syntaxes for calling methods. Some of the presented techniques look like Method Chaining and Custom Return Objects. In yet unpublished work (http://martinfowler.com/dslwip), Fowler provides some interesting techniques for implementing internal DSLs that go beyond a particular dynamic language. It is interesting to further check our found patterns with

other dynamic languages or even static languages to see how they fare with respect to generalization.

Looking explicitly for pattern catalogs in the context of DSL, some examples can be found. In [42], eight general design principles are considered grouped into creational, structural, and behavioral. Creational patterns for example elaborate the principle availability of using a given language for specialization or restriction of its host, as well as using the string processing capabilities of a given language for very simple DSLs. Behavioral pattern explain how to link the output of one DSL with another one using Unix pipes. We see these patterns as providing general implementation schemes while our patterns are detailing how to actually change the language. Another paper explains patterns that govern the principal DSL development process (is the DSL created from scratch or is it based on the "language" of an existing system?), the concrete syntax style (is it a textual or a graphical DSL?), and whether to build an external and internal DSL. The patterns explore the general available design space for DSLs and their influence on each other [51]. Another paper explains patterns for different language engineering concerns. Some patterns express the need to start a language with a minimal feature set, growing it over the time with more functionality. Syntax is of great importance and should be put first as a design criteria. Once the language is bigger, modules in independent files should be used to facilitate maintaining and continuing development with multiple developers. Also, a standard library should be given to support the language's core functionality, and explicit extension points can help to focus the language's evolution [23].

10. Summary

In this paper we presented a rich pattern language describing a principled approach for designing internal DSLs. Based on the defining characteristics of a DSL the following four principles have been presented: *representation*, *absorption*, *compression*, and *generalization*. We explained each design principle with motivation, forces, consequences, and showed an example in which several patterns are used. These principles are concerned with the DSL's syntax, and thus we explained patterns that enable or modify syntactic constructs. We identified three groups of patterns: Form, modification, and support. The form patterns provide the language's overall layout, a broad choice between closely grouped singular statements that can express the natural hierarchy in the domain, or verbose and complex statements that look like language keywords. The modification patterns change the existing language by aliasing entities and module. Finally, the support patterns provide additional syntactic changes that can be combined with the other patterns. We explained each patterns with the classical pattern description form, including the implementation and the known uses. The patterns were illustrated with Ruby, but we also provided known uses for Python.

The final discussion contrasted the patterns from each other and explains how they can be connected. Both the given examples and the known uses show that the explained patterns are ideally used in combination with each other, and allow to build DSLs with a distinct form that do not resemble their host-language's object-oriented nature.

Acknowledgements

We thank the shepherd for helpful comments on an earlier draft of this paper. We also wish to thank Andy Kellens for sharing his insights of the Smalltalk language, and Alexandre Bergel for his input on the presence and use of these patterns in Smalltalk DSLs.

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. Thomas Cleenewerk works with the Language Engineering Lab at the Vrije Universiteit Brussel.

References

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

[2] 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.

[3] 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.

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

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

[6] A. P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Pharo by Example*. Square Bracket Associates,, 2009.

[7] A. P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Squeak by Example*. Square Bracket Associates,, 1st revised edition, 2009.

[8] G. Booch. *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[9] D. Bruce. What makes a good Domain-Specific Language? APOSTLE, and its Approach to Parallel Discrete Event Simulation. In *First ACM SIGPLAN Workshop on Domain-Specific Languages (DSL)*, pages 17–35, Paris, France, 1997. University of Illinois Computer Science.

[10] B. Cannon and E. Wohlstadter. Controlling Access to Resources within the Python Interpreter. *Proceedings of the Second EECE*, 2007.

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

[12] F. F. Correia, H. S. Ferreira, N. Flores, and A. Aguiar. Patterns for Consistent Software Documentation. In *Proceedings of the 16th Conference for Pattern Langauges of Programs (PloP)*.

[13] G. Costagliola, A. Delucia, S. Orefice, and G. Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages & Computing*, 13(6):573–600, 2002.

[14] 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.

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

[16] 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.

[17] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O-Reilly Media, Sebastopol, 2008.

[18] J. Y. Foote B. Metadata and active object models. In *Proceedings of Plop98*, Washington, USA, October 1998. Washington University Department of Computer Science.

[19] 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.

[20] J. F. Groote, S. F. M. Van Vlijmen, and J. W. C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security (COMPASS '95)*, pages 57–68. IEEE, 1995.

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

[22] 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.

[23] A. Haase. Patterns for the Definition of Programming Languages. In *Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2007.

[24] E. W. Høst and B. M. Østvold. The java programmer's phrase book. In *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 322–341, Berlin, Heidelberg, 2008. Springer.

[25] 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.

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

[27] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *VHLLS'94: Proceedings of the USENIX 1994 Very High Level Languages Symposium Proceedings on USENIX 1994 Very High Level Languages Symposium Proceedings*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

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

[29] F. Latry, J. Mercadal, and C. Consel. Staging Telephony Service Creation: A Language Approach. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications (IPTComm)*, pages 99–110, New York, 2007. ACM.

[30] 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, Germany, 2001. Springer Verlag.

[31] M. Lutz. *Learning Python*. O'Reilly Media, Sebastopol, 4th edition, 2009.

[32] W. M. McKeeman. Programming language design. In *Compiler Construction, An Advanced Course, 2nd ed.*, pages 514–524, London, UK, 1976. Springer-Verlag.

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

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

[35] J. Munnelly and S. Clarke. A Domain-Specific Language for Ubiquitous Healthcare. In *Proceedings of the 3rd International Conference on Pervasive Computing and Applications (ICSPA)*, pages 757–762. IEEE, 2008.

[36] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, Mountain View, California, USA, 2008.

[37] J. F. Pane. *A Programming System for Children that is Designed for Usability*. Dissertation, Carnegie Mellon University, Computer Science Department, 2004.

[38] P. Perrotta. *Metaprogramming Ruby*. The Pragmatic Bookshelf, Raleigh, 2010.

[39] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Reading, Harlow et al., 1999.

[40] A. Sharp. *Smalltalk by Example*. McGraw-Hill Publishing Co., 1997.

[41] D. Spiewak and T. Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In *Proceedings of the 2nd International Conference on Software Language Engineering*, Denver, USA, 2009.

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

[43] M. Summerfield. *Programming in Python 3: A Complete Introduction to the Python Programming Language*. Addison-Wesley, Upper Saddle River, Boston et al., 2nd edition, 2010.

[44] É. Tanter. Contextual Values. In *Proceedings of the 2008 Symposium on Dynamic Languages*

(DLS). ACM, 2008.

[45] S. Thibault, R. Marlet, and C. Consel. A Domain-Specific Language for Video Device Drivers: from Design to Implementation. pages 11–26, 1997.

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

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

[48] D. Wampler and A. Payne. *Programming Scala*. O'Reilly Media, Sebastopol, 2009.

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

[50] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

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