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

Sebastian Günther

A great part of software development challenges can be solved by one universal tool: Abstraction. Developers solve development challenges by using expressions and concepts that abstract from too technical details. One especially supportive tool for abstraction are domain-specific languages (DSLs). DSLs are specifically crafted languages that support abstraction in two ways. On the one side, they abstract from the (real-world) concepts of the domain by introducing objects with structure and behavior that are representative for the domain. On the other side, they abstract from a programming language and do not depend on hardware-specific details. We focus on internal DSLs, which are build on top of an existing host language. Such DSLs can completely use the development tools of their host, are quick to build because they reuse existing abstractions, and can be easily integrated with other programs, libraries, and DSLs of the same host.
Developing DSLs is challenging. Especially dynamic object-oriented programming languages like Ruby and Python are used as hosts for DSLs. In these languages, DSLs exploit syntactic variations of the host to express domain-specific notations, and they access the host's reflective facilities and metaprogramming capabilities to implement the domain semantics. Although there are several case studies for DSLs, explanation of a DSLs design and implementation uses mostly host-specific terminology. Here we identify two particular challenge. First, the DSLs design intent is explained too specific to the particular DSL, making it difficult to reuse explained techniques to develop DSLs in another domain. Second, because the DSL design is explained with techniques native to the used host language, it is difficult to transfer these techniques to other host languages. In the long term, this could lead to fragmented islands of DSL design knowledge.
Our solution to these challenges is to provide a host-language independent catalog of design principles and design patterns for DSLs. Particularly, this paper contributes six design principles and nine notation patterns for DSLs. The patterns have been found in 12 DSLs of the three host languages Ruby, Python, and Scala.

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

General Terms: Languages

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

ACM Reference Format:

Günther, S. 2011. Development of Internal Domain-Specific Languages: Design Principles and Design Patterns. 18th Conference on Pattern Languages of Programs - Portland, Oregon, USA (October 2011). 25 pages.

1. INTRODUCTION

Domain-specific languages (DSLs) are tailored towards a specific application area [39]. They use appropriate abstractions and notations to represent the domain [40]. Abstractions are the essence why DSLs support software

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

development concerns. Software development means to formulate the concepts of the problem space (domain model) as concepts of the solution space (application components) [14]. In the problem space, domain concepts and their relationships are represented as a domain model. In the solution space, domain concepts are typically represented as objects and methods of a programming language, or other suitable constructs. This facilitates to test, build, and deploy applications. DSLs close the gap between the problem space and the solution space by providing abstractions that bridge both spaces (see ►Figure 1).

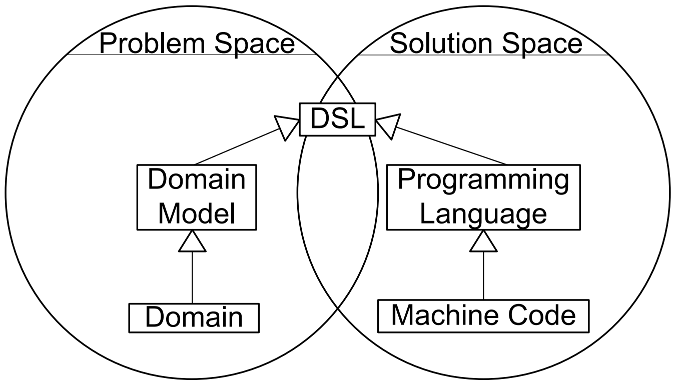

Figure 1: DSL's are abstractions from the problem space and the solution space simultaneously.

There are two kinds of DSLs: external and internal ones. *External DSLs* require building the language from scratch, with custom syntax and semantics. This comes at the cost of implementing the needed compilers/interpreter and other tools by oneself or with the help of a language development environment. On the contrary, *internal DSLs* are built on top of an existing programming language, also called the host language. This allows to use the complete existing language infrastructure [27], including the compiler or interpreter, debugger, and IDE's.

In general, DSLs should be developed only if they are expected to have a profound impact on development productivity and are reused in future developments [27]. The question whether an internal DSL or an external DSL should be developed depends on factors such as the skills of the developers, the complexity of the domain, and the platform where this language is used, especially its integration with other libraries or tools. We argue that the development of external DSLs is more heavyweight because the language needs to be completely specified (including abstract syntax, concrete syntax, and (often) translational semantics) without giving the designer a familiar set of abstractions. Moreover, the maintenance effort for an external DSL because the generated code needs to be adapted to the specifics of the platform and surrounding applications in which it is going to be used. In contrast, internal DSLs are more lightweight: The host offers familiar abstractions that developers can leverage, the DSL itself is compatible with the host-language and can be easily integrated with other libraries or DSLs, and the development and maintenance effort is low because the developer' familiar processes and tools can be used. For these reasons, we focus in this paper on internal DSLs.

Several case studies for internal DSLs exist. However, we see two particular challenges arising. First, most case studies have a strong influence of the particular domain that they use and the explanation how it was implemented. Too technical considerations mask the intent, the goal behind using a particular notation or abstraction. This makes it difficult to use the techniques for another domain. Second, because the explained techniques are host-language dependent, they use host-specific vocabulary, which is difficult to transfer to another host language. In the long term, this could lead to isolated DSL development knowledge in different communities.

Our answer to these challenges is to discover the essential DSL design principles and DSL design patterns. Design principles are guidelines that explain how to approach a DSL design to manifest specific properties in the

DSL. Design patterns identify common challenges in the design of notations and abstractions for a DSL. They explain the design considerations, and discuss potential solutions.

Although related work also explains design principles and design patterns, there remains a large knowledge gap:

– *DSL design principles*: DSLs have unique properties that are difficult to express as principles for guiding the DSL development. Most internal and external DSL case studies such as [19][37][3][1][15][4][23] do not discuss design principles or explain their DSL's design properties. Existing work on principles uses different names and explanations for similar concepts. For example, there are principles about designing notations for visual constructs [11][29], for building abstractions [24], for programming language design [28], or specifically for DSLs [30]. We propose a set of principles that are originating from the definition of DSLs, and we analyze and consolidate the principles mentioned in other works. These principles form a coherent vocabulary to guide the design of DSLs.

– *DSL design patterns*: The actual implementation of a DSL on top of a host language requires careful design choices. Most case studies do not generalize their approach and mix host language characteristics, as well as language constructs, metaprogramming and design considerations. More concretely, the case studies suggest to use host language specific metaprogramming methods [13], internal interpreter objects [15][33], object-oriented programming constructs, in particular methods [8][1], or even the host's metaobject protocol for implementing a DSL [15][12]. We provide a thorough analysis and description of the different mechanisms in the form of patterns. Our patterns record the context, problem, and solution of DSL design challenges and are used in DSLs based on Ruby, Python, and Scala.

For these reasons, this paper provides a list of essential design principles and design patterns. Specifically, we explain the six principles notation, compressions, and absorption as well as abstraction, generalization, and optimization. For the design patterns, we explain the essential notation patterns, grouped into layout patterns (Provide the general layout of the DSLs), expression patterns (simplify and straighten the representation of domain concerns), and support patterns (unique effects to simplify expressions.)

In the remainder of this paper, we start to explain the particular pattern structure in chapter 2. Then, chapter 3 explains the design principles and chapter 4 lists the design patterns. Chapter 5 discusses the pattern relationships, and chapter 6 summarizes the paper.

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

2. PATTERN STRUCTURE

The pattern explanation has a structure that is close to the initial introduction of patterns [18] and recent publications [20][10][44]. To show the applicability of the patterns, we research how they can be utilized in three different host languages. We select the two dynamic host languages Ruby and Python because of their popularity as listed in the Tiobe language popularity index[1] and the large number of DSL projects as listed in Github[2]. As a contrast, we also use the compiled programming language Scala which is not so popular on Tiobe, but has several DSL projects listed on Github, and even includes a DSL in its core contribution.

2.1 Pattern Structure

The following structure is used to explain each pattern:

– *Name*: The name of the pattern.

[1] http://tiobe.com
[2] http://github.com

– *Also Known As*: After we finished our research, another research work [17] was published in which patterns similar to our are published. If patterns of our catalog and of [17] are related, we name it to the beginning of each pattern explanation.
– *Context*: Explains a particular design or implementation scenario in which the application of the pattern is beneficial (note that the context follows right after the name, without an additional heading).
– *Problem*: A question that raises a common challenge in DSLs, and an explanation of how the challenge negatively effects the DSL.
– *Solution*: A general advice what to do, followed by a source code example illustrating the pattern's application, and then a detailed explanation of the solution and its host-language specific implications.
– *Design Goals*: Explains explicit design goals and DSL design principles that are achieved by applying this pattern. Usually, one pattern can be used to support different principles. In some cases, one pattern can be used in two distinctive ways to support a principle, which we document both.
– *Liabilities*: Explains potential pitfalls and tradeoffs when this pattern is applied.
– *Known Uses*: We document at least one occurrence of this pattern as long as we found it being used in the analyzed DSLs.
– *Related Patterns*: Finally we explain how this pattern can be combined with other patterns.

2.2 Analyzed DSLs for Known Uses

We analyze DSLs for web applications, test-driven development, cryptography, load balancing, and others.

- *Ruby*
  - *Rails* (version 2.3.5): Web-application framework with a strong Model-View-Controller architecture that provides many convenient expressions as a DSL for web applications. Rails is used in several open-source and commercial projects. Rails is closely bundled with other libraries that we also analyzed: *ActiveSupport* (version 2.3.5) adds helper classes, *Actionpack* (version 2.3.5) provides request and response objects, and *Builder* (version 2.1.2) is used to build configuration objects representing XML-like data structures (http://rubyonrails.com).
  - *ActiveRecord* (version 2.3.5): Provides database abstraction with the philosophy that an object encapsulates the data of one specific database row. Also it can be used as a standalone DSL, it is often bundled with Rails (http://ar.rubyonrails.org/).
  - *RSpec* (version 1.3.0): Framework and DSL for behavior-driven development (http://rspec.info).
  - *Sinatra* (version 0.9.4): Lightweight web framework and DSL that uses declarative expressions for specifying request handling (http://sinatrarb.com).
- *Python*
  - *Bottle* (version 0.8.3): Similar to Sinatra, a lightweight web framework that uses normal Python function declarations and annotations to specify how queries are responded to in the application (http://bottlepy.org).
  - *Should DSL* (version 1.0[3]): Supports simple behavior-driven testing by adding expressions like `should`, `be`, and `be_greater_then` as expressions in Python (http://github.com/hugobr/should-dsl).
  - *Cryptlang*[4]: A DSL for defining cryptographic algorithms and random number generators [1].
  - *LCS DSL*[4]: The Constellation Launch Control System DSL, or short LCS DSL, provides a command like language for "programming test, checkout, and launch processing applications for flight and ground systems" [4].
- *Scala*

[3]The version 1.0 was submitted on June 17th, 2009, we use the more recent commit with the id 1725b62ad30195f578a4, issued on July 15th, 2010.
[4]For these DSLs, we do not have access to the source code, and can only show the pattern application, but not explain their declaration.

– *Actors* (version 2.7.7final): Actors is a built-in Scala DSL for asynchronous messaging, and frequently deployed for concurrent programming. It uses concepts like a messaging box and special operators to send and retrieve messages (http://www.scala-lang.org/api/current/scala/actors/package.html).
– *Specs* (version 1.6.2.1): DSL for behavior driven development similar to the block-like nature of RSpec (http://code.google.com/p/specs/).
– *Apache Camel* (version 1.6.4): Helps to express the configuration of enterprise systems that consist of Java-compliant applications, allowing to define routes and APIs for the components (http://camel.apache.org/scala-dsl.html).
– *ScalaQL*[5]: Implementation of SQL as a Scala DSL [34].

2.3 Common Example

The common example that we use to explain the patterns is a DSL that we created to model software product lines [21]. A product line helps to structure common and variable assets to describe a family of related products [14]. Assets can be of functional nature, such as source code, binaries, and libraries, or they can in other ways contribute to the program, like help files, graphics, or other media files. The product line allows to configure a concrete product by determining its features. Features are functional properties of the application [43], and typically they are represented as feature trees [14]. In these trees, the natural relationships and constraints between features are represented, where the selection of one feature can mean the inclusion, exclusion, or choice of other features.

Our example is a textual DSL for these feature models. We want to define the features of a car product line. This product line has a `Car` feature at the root and three subfeatures: `Colors`, `Transmission`, and `Options`. The features `Colors` and `Transmission` are mandatory, while `Options` are optionally included in the product line. To express the `Car` feature, we use the following DSL expression:

```
1 car = Feature.configure do
2   name "Car"
3   root
4   subfeatures :Colors, :Transmission, :Options
5   require :car => "all :Colors, :Transmission"
6   require :car => "any :Options"
7 end
```

From top to bottom, these expressions mean the following:

– *Line 1*: `configure` is a method that receives a number of configuration option with which the created feature object is configured.

– *Line 2*: The internal `name` that is used to identify this feature.

– *Line 3*: `root` expresses that this particular feature form the root of the feature tree.

– *Line 4*: `subfeatures` build the visual structure of the tree by naming all subfeatures of `Car`.

– *Line 5–6*: A complex expression that determines the constraints of Car to its subfeatures. The first expression means that when `Car` is selected, then also the features `Colors` and `Transmission` have to be selected in the configuration. The second expression means that the `Options` feature can be chosen optionally.

3. DESIGN PRINCIPLES

Design principles are guidelines that explain how to approach a DSL design to manifest specific properties in the DSL. We explain six principles. The first group of principles manifests the DSL's semantic properties (abstraction, generalization, and optimization), and the second group manifests the DSL's syntactic properties (notation, absorption, and compression). Each principle is detailed in the following paragraphs.

Abstraction

Goal: Implement domain concepts as semantic abstractions that are not too tied to host-language semantics.

DSLs abstract twofold: From the domain and from their host language. Abstracting from the domain is to include only those entities and operations in the language that are relevant. This means to provide "crisply defined conceptual boundaries" [6] to find entities that "fit the domain closely" [2]. Abstraction from the host language means to implement these domain concepts in an operational model that is not too tied to the existing host language semantics.

Generalization

Goal: Choose a small amount of concepts for the DSL that are derived from the greater hierarchy of domain concepts.

Generalization is an important concept in domain engineering and in software engineering. First, the domain is captured in its full scope consisting of multiple concepts and their relationships. They naturally form a hierarchy in which one concept is more general than other specific concepts. Finding such hierarchies helps the developer to understand the domain. Generalization keeps DSLs small as the reduction of the amount of concepts results in a small amount of needed constructs [9] – the developer needs to choose how to prune, reorganize, and combine the concepts.

Optimization

Goal: Implement structural and algorithmic optimizations to improve the DSL's computational performance.

Producing efficient code is an essential characteristic of programming languages. According to [30], DSLs allow two forms of optimization. The first form is to optimize how the DSL code is executed as host language code, for example thorough directed memory allocation. The second form is to implement algorithms in such a way that they are optimal for the represented domain, which means to use suitable models and domain knowledge for designing an efficient implementation [7].

Notation

Goal: Represent domain-specific concepts with concise and suitable entities.

A DSL's syntax is the sum of the host's syntax and the domain-specific notations. The syntax consists of all language constructs, keywords, structures, and the rules governing their combination. The rules determine the general layout of the language. Thereby, a "trade-off between naturalness of expression and ease of implementation" [5] has to be made. Internal DSLs cannot leave the syntactic frontiers of their host language, but they can chose to combine existing syntactic modifications to achieve a form that has little or no resemblance to the host language. Providing the appropriate notations [40] through defining clear distinguishable and representative entities defines the DSL's syntax.

Compression

Goal: Remove unneeded tokens or keywords from the DSL to improve conciseness.

The goal of compression is to provide a concise language that is sufficiently verbose for the domain experts. By reducing the amount of expressions or by simplifying their appearance while the semantics are not changed, the code footprint of DSL programs is reduced. This leads to better understanding the application language and the domain [25][42].

Absorption

Goal: Replace explicit concerns with implicit assumptions that are communicated through DSL expressions

Certain characteristic concerns of domains are common yet explicitly represented in DSL expressions. These concerns can be pulled into the behavior of domain operations, and by finding suitable expressions they can be communicated implicitly. High absorption facilitates using the DSL because repetitive expressions are removed, it provides DSLs implicit focused expressive power [36].

4. NOTATION PATTERNS

Notation patterns capture a particular design intent to provide domain-specific notations. Three groups of notation patterns can be found. *Layout patterns* provide the overall structure of the DSL. *Expression patterns* provide syntactic alternatives for expressions, especially for method calls, in order for a more domain-specific notation. *Support patterns* further help to provide clearer expressions. We explain all patterns in the following sections.

4.1 Layout Patterns

Layout patterns determine a DSL's overall layout. The primary choice is between a horizontal and a vertical code layout. Vertical code layouts emphasize the importance of blocks of code and small, concise expressions. This is especially useful to represent the hierarchies of domain concepts. In contrast, horizontal code layouts provide more complex, but very readable expressions, relating whole collections of expressions to each other. Layout patterns are BLOCK SCOPE, METHOD CHAINING, and KEYWORD ARGUMENTS.

4.1.1 *Block Scope*

Also known as Nested Closure [17].

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

Problem

Has your DSL a flat scope where you mangle objects together? Or is it difficult to express hierarchies of domain objects? A (too) flat scope hides the context and natural connections between expressions, masking their intent.

Solution

Group related expressions together in one block, and define a method receiving and executing this block.

```
car_feature = Feature.new(:Car)
car.feature.root
car.feature.subfeatures(:Colors, :Transmission, :Options)
```

$\downarrow$

```
car = Feature.configure do
  name "Car"
  root
  subfeatures :colors, :transmission, :options
end
```

Syntactically, a BLOCK SCOPE is defined by the visual arrangement of expressions as indented lines of code. Semantically, the inner part of the block absorbs execution information from the surrounding scope at creation time, but at execution time, parts of the surroundings can be replaced. Here are the example expressions for each host language:

```
#Ruby
Feature "Car" do
  root
```

```
end

#Python
with Feature("Car") as car_feature:
    car_feature.root()

#Scala
Feature("Car") configure {
  root()
}
```

In Ruby, blocks are build by using either curly braces or the above shown `do...end` notation. The code inside the block captures variable bindings from the surroundings, so it can be executed independently of its original surroundings. If the block is later executed in another scope where similar variables are defined, these variables override the original bindings.

Python can use a block-like structure with context managers. A context manager is any objects that defines the methods `__enter__` and `__exit__` which are executed at the same-named time during the blocks creation and usage. Then, this object can become part of the above shown `with` expression, where the `__enter__` method is executed and its return value bound to the provided variable reference. In contrast to Ruby, method calls are not implicitly called, but still need to provide the method receiver.

Scala uses curly braces as the syntax for introducing blocks. Blocks are not special data types and they do not share Ruby's closure capabilities, they are a way to group related expressions. Technically, when using such blocks, the block is first executed and its return value returned to the function that opens the block.

Design Goals

– *Notation*: Provide a vertical code layout to group related expressions in an indented block of code.
– *Absorption*: Form a closure that absorbs the surrounding context information into a block of code.
– *Compression*: In the block of code, replace fully-qualified method calls with local ones, and then let the execution environment of the block determine the concrete semantics.
– *Abstraction*: Change the semantics of the block by changing the execution environment of the code.

Liabilities

– Possibility of code injection: When blocks are dynamically constructed from user input, then code could be ejected to perform e.g. operations that read the application or access the file system where the application is mounted. There are host-language specific options to prevent malicious code from being executed, for example does Ruby support safe levels that determine what the executed code can do (for example preventing to access the file system) [38].

Known Uses

– Ruby
  – Using *Sinatra*, BLOCK SCOPE is applied with the declaration of request handlers, for example `get`, to obtain a declarative expression how a web application reacts to queried URLs (the `get` request handler is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Lines 752–758).
  – Using *RSpec*, BLOCK SCOPE is applied for the declaration of example groups that begin with the `describe` keyword and group several test cases (the `describe` method is implemented in rspec-1.3.0/lib/spec/dsl/main.rb, Lines 24–29).
– Python
  – We could not find this pattern in the analyzed DSLs. In our PyQL DSL [22], we use the context manager construct to introduce a block. The expression `with Query("sqlite") as query` starts a block where

`query` is a local variable. Using this object, methods representing SQL-statements are called, but fully-qualified method calls have to be used.

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

Related Patterns

– KEYWORD ARGUMENTS: If a horizontal code layout is required, then KEYWORD ARGUMENTS can be used.

4.1.2 *Method Chaining*

Also known as Method Chaining [17].

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

Problem

Do you use too many expressions line-per-line that do not relate domain concept appropriately? Do you need verbose expressions for a particular complex domain intent? Too many command-like expressions make the DSL hard to understand.

Solution

Provide a readable expression where method calls are chained together in one line of code.

```
car.feature.add_subfeature :Colors
```

$$\downarrow$$

```
add subfeature Colors to "Car"
```

METHOD CHAINING has two operational forms: Either the method calls manipulate shared or global data, or data is explicitly passed between method calls. The later often requires a rather extensive implementation that depends on the length of the method chain.

Methods in METHOD CHAINING usually take one of these roles:

– *Context provider*: The first-called method provides the data that is manipulated.
– *Referrer*: Methods that act as placeholder for a global visible data.
– *Glue*: Methods that serve to improve the readability of the expressions without actual semantics.

METHOD CHAINING takes very different forms dependent on the used host language. Ruby allows to call methods without using the typical "dot notation", forming very concise expressions. In contrast, Python expressions must use the dot notation and parentheses on method calls. In Scala, it is especially tricky to form chains because an alternating combination of object references and method calls must be used.

```
add subfeature Colors to "Car"         #Ruby
add().subfeature(Colors).to("Car")     #Python
add Colors to feature("Car")           #Scala
```

Design Goals

– *Notation*: Improve the syntactic conciseness by chaining method calls and their return values together. This also allows you to change the code layout of the DSL to a horizontal representation.
– *Compression*: Adopt a functional-programming like style by directly passing arguments between method calls, to transform an input value step-by-step.

Liabilities

– Too many glue methods can pollute the namespace of the DSL.
– It is hard to ensure the correct order of calling methods in the chain, implementing checks in each method can be very cumbersome.
– Limited applicability to external libraries because of the high implementation effort to design the chain.

Known Uses

– Ruby
  – In *RSpec*, users chain `should` and `should_not` expression with an expectations statement like `should_not be_nil` or `should have(3).items`, combining the statements in a clean and readable way (both mentioned methods are defined in rspec-1.3.0/lib/spec/expectations/extensions/kernel.rb, Lines 26–28 and Lines 49–51, which extends Ruby's built-in `Kernel` module that defines global available methods).
– Python
  – The *Should DSL* is a unique extension because it uses OPERATOR EXPRESSIONS (explained later) together with a normal method call to add a new syntactic element to Python. Basically, it allows the same kinds of expressions like RSpec, consisting of an object, a `should` keyword, and an expectation. Here is a short example: `1 |should_not| be(2)`. Inside this expression, the "`|`" are actually method calls that receive the left value and the right value, which is then passed to the `should_not` method. This combination allows METHOD CHAINING (the "`|`" operators are defined in should-dsl-1.0/src/should_dsl.py, Lines 25–38).
  – The *LCS DSL* also uses a form of METHOD CHAINING: Methods with boolean return values are used to structure conditional blocks that determine how to react to different status of the monitored applications [4] (we can not give further details because we do not have access to this DSL's implementation).
– Scala
  – *Specs* uses method chaining in an extreme form, because expressions can mirror complete sentences. For example we create a car object with the name "Nissan Primera". To test whether the car name begins with "Niss" and not "Pors", this expression can be used: `nissan.carname must be matching("Niss*") and not be matching("Pors*")`.
  Similar matchers for non-string objects are included in the Specs library, or can be added for user-specific types (the `must` method is defined in specs-1.6.2.1/src/main/scala/org/specs/specification/Expectable.scala, Line 172, and the matchers `be` and `matching` are defined in specs-1.6.2.1/src/main/scala/org/specs/matcher/StringMatchers.scala, Line 36 and 202).

Related Patterns

– KEYWORD ARGUMENTS: Horizontal code layouts are provided with METHOD CHAINING and KEYWORD ARGUMENTS, where KEYWORD ARGUMENTS can be used to refactor too complex expressions into subphrases. This also removes the need to use glue methods.
– BLOCK SCOPE: The BLOCK SCOPE pattern should be used as an alternative especially when hierarchies of domain concepts need to be expressed. Also, inside a BLOCK SCOPE, individual lines can be expressed with METHOD CHAINING.

4.1.3 *Keyword Arguments*

Also known as Literal Map [17].

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

Problem

Do your DSL functions receive too many arguments, where the order or meaning of the arguments are hard to remember? When hard to track errors occur because arguments are passed on the wrong order, this can frustrate the user.

Solution

Use key-value pairs where the key defines the arguments meaning and the value is the arguments value.

```
car = Feature.configure do
  name "Car"
  root
end
```

$\downarrow$

```
Feature.configure :name => "Car", :position => "Root"
```

The notion of key-value pairs, usually as a Hash, is a shared construct by many host languages. Some languages even allow to split parts of key-value pairs along several source-code lines, proving a source code layout similar to BLOCK SCOPE. The host languages can use the following notations for using the KEYWORD ARGUMENTS pattern:

```
feature :name => "Car", :position => "root"    #Ruby
feature(name="Car", position="Root")           #Python
feature("name" -> "car", "position" -> "Root") #Scala
```

As can be seen, key-value pairs are defined in Ruby as `Hashes` with `key => value`, Python uses the internal type `dict`, and in Scala, `Tuples` are used.

Design Goals

– *Notation*: Refactor long argument lists to method calls by clarifying the meaning of each argument and improve the flexibility of the argument order.
– *Notation*: Express relationships between arguments by using appropriate keyword-value pairs.
– *Absorption*: Use keyword-value pairs to infer implicit meaning of the method call.
– *Compression*: Replace single method calls with one method calls that uses key-value pairs.
– *Generalization*: The methods behavior is determined by optionally passed arguments, providing a semantic focus for adding new behavior to the DSL.

Liabilities

– Avoid overly complex key-value pairs that have too many semantic meanings dependent on their existence – in this case, refactor to different methods.

Known Uses

– Ruby

– *Sinatra* uses the `set` method for configuring options, it receives a single key-value pair and sets the configuration accordingly (the `set` method is implemented in sinatra-0.9.4/lib/sinatra/base.rb, Lines 622–636).

– *ActiveRecord* uses the built-in `autoload` method that receives a module name as a symbol and a filename in which the module is implemented. Modules configured in this way are only imported when they are called in the program (the `autoload` method is defined in Ruby's core class `Module`, which is part of the interpreter implementation).

– Python

– In *Bottle*, request handlers are defined with normal Python functions and annotation. The annotations determine which queried URL triggers the execution of a method. The annotation can contain additional keywords, for example `@route('/store/product/:id', method='POST')`. This defines a HTTP POST request with the specified URLs to trigger the annotated method (the `route` method is defined in bottle-0.8.4/lib/bottle.py, Lines 398–443).

– The *LCS DSL* uses a variant of keyword passing. Instead of using hash structures directly, first strings are passed to the functions and then the values of the variables addressed by this string. Here is an example: `send_command (discrete ("VALVE1", "ON"))` [4] (we cannot give further details because we do not have access to the implementation).

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

Related Patterns

– Block Scope: Where key-value pairs provide a horizontal code layout, Block Scope can express the same meaning with individual lines of code grouped in a block, but the keys of the key-value pairs need to be implemented as methods.

– Method Chaining: Simple key-value pairs can be represented with Method Chaining too, but again additional methods have to be implemented.

4.2 Expression Patterns

Expression patterns focus on the design of individual expressions in the DSL. They are concerned to form unique expressions that clearly represent the domain, and therefore reduce for example the non domain-related tokens or keywords from the expressions, or replace usage of host language built-in entities with notations from the DSL.

4.2.1 *Seamless Constructor*

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, domain-specific meaning. More desirable is the behavior that an expression automatically instantiates an object, for example executing "`42`" returns an integer object.

Problem

How to replace explicit object instantiation with expressions more suitable for the domain? The mechanism or keyword to create new instances could be confused with a domain keyword, or it could impair the readability of the DSL.

Solution

Create instances as implicit side effect of DSL expressions.

```
car_feature = Feature.new("Car")
```

$$\downarrow$$

```
car_feature = Feature("Car")
```

The most easy solution is to put the creation of instances as side effects of specific DSL expressions. Another option is to use the same object for this expression, but exploiting lexical scoping to define a same-named method that creates instances.

In Ruby, methods with same symbols (including capitalization) as existing classes can be used in those places in the source code where method calls are expected. We used this particular property in the above example. Python uses this pattern as the standard way to create instances: `Feature()` create a new instance of this class. In Scala, either case classes are used to define simple data containers that are instantiated without using an explicit keyword like `new`, or similar to Ruby, the same symbol of the class mat be used to define a method that does this instantiation.

Design Goals

– *Notation*: Improve the DSL's readability by removing explicit object instantiation from the DSL.
– *Absorption*: Create instances implicitly through suitable expressions instead of using explicit keywords.

Liabilities

– Using a SEAMLESS CONSTRUCTOR that has the same symbol as the class for which it creates an instance is limited to valid positions according to the grammatical rules of the host language.
– In some host languages, using method names with capital letters violates coding conventions.

Known Uses

– In Ruby, no example could be found in the other analyzed DSLs.
– For Python, the *Should DSL* defines custom variables that point to object constructors. The method `should` actually calls the constructor `Should()` to create a new instance (the declaration of the `should` method is contained in should-dsl-1.0/src/should_dsl.py, Line 224).
– For Scala, this pattern could not be found in the analyzed DSLs.

Related Patterns

– ENTITY ALIAS: Instead of hiding the intent to create instances completely, the specific keyword or class can be replaced with a notation that better expresses the domain.

4.2.2 *Superscope*

DSL expressions often need to relate several objects in one operation. These objects, however, may not all be defined in the same namespace in which the expression is formed. One solution is to introduce variables in the local scope that refer to the used objects, but this means to clutter the namespace. We need a mechanism that allows to refer to objects of any scope, without worrying on the caller side where the referenced objects actually are.

Problem

How to add references to objects in another scope without using fixed or bound variables? Fixed variables limit the DSL's capabilities to offer several namespaces where DSL expressions can be defined.

Solution

Define a suitable replacement for the object reference, such as a string or symbol, and implement a mechanism to resolve the reference to the concrete object.

```
car_feature = Feature.configure do
  # All subfeatures have to be defined in the same scope.
  subfeatures Colors, Transmission, Options
end
```

$$\downarrow$$

```
car_feature = Feature.configure do
  subfeatures :Colors, :Transmission, :Options
end
```

This pattern can be implemented with basic host language mechanisms. The reference is either a string, a symbol, or a similar object. The mechanism that resolves the reference to the concrete object can be a method that has stored the namespace-specific and just parses the passed reference to return the concrete object.

Design Goals

– *Notation*: Eliminate the need to prefix object references with namespace details.
– *Compression*: Shorten object references.
– *Generalization*: Use indirect pointers instead of fixed references to change the concrete objects as needed.

Liabilities

– The reference resolving mechanisms hardcodes the namespace details and must be maintained when the namespace of objects changes.
– When arguments uses as SUPERSCOPE come from user input, this could be exploited to access inappropriate objects.

Known Uses

– Ruby
  – In *Rails*, `redirect_to :controller => "registration", :action => "confirm"` is an expression inside handler declarations to construct a custom URL that is returned to the request. The expression uses `Superscope` to denote the controller and the called action. The advantage of this expression, compared to the static configuration of a route, is that the arguments are checked to actually exist and that further options, such as the status code, can be added (`redirect_to` is defined in actionpack-2.3.5/lib/action_controller/base.rb, Lines 1103–1133).
– Python
  – The *LCS DSL* uses SUPERSCOPE to refer to DOMAIN OBJECTS and DOMAIN OPERATIONS in the expression `send_command (open ("VALVE1", "ON) )` [4] (we cannot give further details because we do not have access to the source code).
– Scala
  – The *Apache Camel DSL* expression `app1 throttle (3 per 2 seconds) to (app2)`[6] uses SUPERSCOPE to refer to the application URLs between which redirections are configured. The URLs are associated with `SRouteType` classes (the method `throttle` and `to` are defined in apache-camel-1.6.4/components/camel-scala/src/main/scala/org/apache/camel/scala/dsl/SAbstractType.scala, Line 85 and Lines 33–42 respectively, and the class `SRouteType` is defined in apache-camel-1.6.4/components/camel-scala/src/main/scala/org/apache/camel/scala/dsl/SRouteType.scala).

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

Related Patterns

– KEYWORD ARGUMENTS: Both key and value can be SUPERSCOPE objects, providing a very flexible way to determine the concrete semantics of expressions.
– METHOD CHAINING: Some expression parts can be replaced with SUPERSCOPE arguments, giving greater flexibility to the scope where the method chain is defined.
– CUSTOM RETURN OBJECTS: Specifically tailor the objects that is referenced by a SUPERSCOPE object.

4.2.3 *Entity Alias*

A programming language offers extensive functionality through its core-library and several external libraries. However, this functionality may be expressed in an unintuitive or misguiding way if it were to be used in a DSL. Using appropriate, domain-specific names for those built-in methods may be crucial for the DSL.

Problem

How to use more domain-suitable notations to use built-in classes or external libraries in DSL expressions? The symbols used for built-in classes or methods may conflict with the meaning that the DSL tries to achieve.

Solution

Use domain-specific objects and methods that refer to the built-in ones.

```
Features.all.to_array
```

$\downarrow$

```
Features.all.to_feature_list
```

Dependent on the type of the referenced object, this pattern is distinguished into *Class Alias*, *Module Alias*, and *Method Alias*. The principal solution is to explicitly define a replacement class, module, or method that just refers to the original one, using the conventional mechanisms to create such objects in Ruby, Python, and Scala. The other solution is to use explicit keywords of the host language, such as `alias` or `alias_method` in Ruby.

Note that additionally to changing the notation that is used to refer to an object, this mechanism can also be exploited to give references to objects that exist in another namespace (similar to what SUPERSCOPE does.)

Design Goals

– *Notation*: Change the name of existing modules, classes, and methods to a better, domain-specific name by providing an alias.
– *Abstraction*: Resolve name clashes when using multiple DSLs.

Liabilities

– Aliases need to be chosen carefully to not conflict with existing, same-named objects.

Known Uses

– Ruby
  – *Rails* has a central generator that is used to provide code stubs for models, controllers, and actions. The generator uses several METHOD ALIAS to provide more convenient method names, for example there is an alias for the `file` command to use `template` (this alias is defined in rails-2.3.5/lib/rails_generator/commands.rb, Line 472).

    – *RSpec* allows to use the name `context` to describe an example group. This method is just an alias for `describe` (the definition of `context` as a *Method Alias* is expressed in rspec-1.3.0/lib/spec/dsl/main.rb, Line 30).
- – Python
    – *Bottle* uses this capability to build decorators: it defines methods such as `request` and `response` that receive a method declaration and use it for defining another method (the mentioned methods are defined in bottle-0.8.3/bottle.py, Line 2120 and 2125).
- – Scala
    – *Specs* implements a mechanism that works like an alias. Normally, the `should` keyword begins a test case. However, using `def configure = addToSusVerb("configure")`, the call can be extended to `should configure`, where `configure` has the same behavior as `should`. However, expectations must always begin with the keyword `should` (the `addToSusVerb` method is defined in specs-1.6.2.1/src/main/scala/org/specs/specification/SpecificationSystems.scala, Lines 71–79).

Related Patterns

– SEAMLESS CONSTRUCTOR: Referenced objects can be used similarly as SEAMLESS CONSTRUCTOR to hide explicit object instantiation commands.

4.2.4 *Operator Expressions*

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. For example, consider arrays where the set operations difference and joining are represented by the symbols "`-`" and "`&`". These symbols are perfectly suited to express the semantics of similar domain operations.

Problem

How can operators, for example mathematical (`+`, `*`) or logical (`&`, `|`) symbols, be used in DSL expressions?

Solution

Define operators as methods in the DSL, using appropriate scoping rules.

```
car_feature.subfeatures :Colors, :Transmission, :Options
```

$\downarrow$

```
car_feature.subfeatures = :Colors + :Transmission + :Options
```

In Ruby and Scala, operator methods are defined through methods that use the operator symbol as its identifier. In Python, operators methods are defined through special keywords. All available operators and their implementation is shown in the ►Table 1 for Ruby (from [16]), ►Table 2 for Python (from [26][35]), and ►Table 3 for Scala 4.2.4 (from [41]).

Design Goals

– *Notation*: Use symbols that intuitively and concisely communicate their intent.
– *Compression*: Replace method names with short symbols.
– *Generalization*: Use operators and operator interaction to provide specialized behavior in the DSL.

Liabilities

– Some operators are required to be defined in other applications or core library classes too.

Table 1 – OPERATOR EXPRESSIONS: available operators in Ruby.

Operator	(Original) 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 2 – OPERATOR EXPRESSIONS: available operators in Python.
Note that for all operations, there is also a right-associative version
available by just prepending a "r" to the methods, for example `__radd__`.

Operator	Method	(Original) Operation
+ -	`__add__`, `__pos__`, `__sub__`, `__neg__`	Addition, substraction (binary and unary)
* **	`__mul__`, `__pow__`	Multiplication, power
%	`__mod__`	Modulo
// /	`__floordiv__`, `__truediv__`	Integer division and float division
& |	`__and__`, `__or__`	Logical "and" and "or"
< > <= >= == !=	`__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`	Comparison operators
« » ~	`__lshift__`, `__rshift__`, `__invert__`	Bitwise shift and invert operations

Table 3 – OPERATOR EXPRESSIONS: available operators in Scala.

Operator	(Original) Operation
=, <,>	Equality and comparison
+, -, *, /	Mathematical operators
\, |	Backslash and bar
!, ?, &, :	Exclamation, question mark, ampersand and colon
%, ^, @, #, ~	Miscellaneous symbols

Known Uses

– Ruby
  – *ActiveRecord* uses "«" to add records to associations (defined in activerecord-2.3.5/lib/active_record/associations/association_collection.rb, Lines 111–125).

– Python
  – *Cryptolang* defines 21 operations to express cryptographic algorithms with a very concise syntax. For example, in the declaration of the blowfish algorithm, one used expression is `y = y ^ (bs[2] $ S[2])`. This expression combines logical-or expressions ("^") and a substitution ("$") by swapping array values [1] (we cannot give further details because we do not have access to the DSL's source code).

– Scala
  – *Actor* 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-2.7.7final/src/actors/scala/actors/Channel.scala, Lines 60–62, Lines 159–165, and Lines 99–101 respectively).

Related Patterns

– ENTITY ALIAS/METHOD ALIAS: Instead of using operators for method names, the alias mechanism can be used to replace the operator with a method name.

4.3 Support Patterns

The support patterns provide additional modifications of the syntax to support all other patterns. Again, they focus to improve the syntax by further removing host-language specific notations that do not have a meaning in the DSL.

4.3.1 *Clean Method Calls*

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 braces. Except structuring more complex expressions, these symbols usually do not add to a DSL's readability.

Problem

How to remove parentheses, semicolon and other expression delimiters that have no meaning of the DSL? Keeping those elements impairs the readability of the DSL expressions.

Solution

Carefully analyze the host-language specific options to remove parentheses for method calls or for putting method calls together.

```ruby
car_feature = Feature.configure do
  require( {:car => "all :Colors, :Transmission"} )
end
```

$\downarrow$

```ruby
car_feature = Feature.configure do
  require :car => "all :Colors, :Transmission"
end
```

How and if this pattern can be used to design a DSL is entirely specific to the host language. Ruby offers a great degree of freedom, for example parentheses can be removed from most method call, there is no need to use curly braces around Hash declarations, lines are delimited by an invisible linebreak character. Python only allows to remove parentheses when OPERATOR EXPRESSIONS are used. In Scala, parentheses in method calls can be removed if there are either no arguments passed or only one argument which is again a method call.

Design Goals

– *Notation* – Gain clear and readable expressions by maximizing the use of domain-specific notations and minimizing the use of redundant host-language expressions.
– *Compression* – Shorten DSL expressions by removing cluttering parentheses, semicolons, and similar expressions.

Liabilities

– The meaning of symbols is determined by the programming language's grammar, and in some cases, parentheses and similar symbols may be required to correctly denote the expressions.

Known Uses

– Ruby
  – In *Sinatra* applications, the request handlers, for example `get`, are typically declared using CLEAN METHOD CALLS (the `get` method is defined in sinatra-0.9.4/lib/sinatra/base.rb, Lines 752–758).

– In *RSpec*, example groups that are introduced with the `describe` keyword are also used without 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, Lines 24–29).

– Python

– The *Should DSL* cleverly uses OPERATOR EXPRESSIONS to enable very readable expressions like `Collection.item_count() |should| be(22)`. Here, "`|`" and `should` are methods (these methods are defined as `__ror__`, `__or__`, and `should` in should-dsl-1.0/lib/should_dsl.py, Lines 25–28, Lines 30–28, and Line 224).

– Scala

– In *Specs*, expressions like `should not match("String")` are available (the `should` method is defined type in specs-1.6.2.1/src/main/scala/org/specs/specification/Sus.scala, for different types in Lines 65–69 and Lines 71–74).

– The *Apache Camel DSL* uses CLEAN METHOD CALLS at several occasions, like in the expression `"app1" throttle (3 per 2 seconds) to ("app2")` which expresses the intent to only push 3 messages per 2 seconds from endpoint to endpoint (the `throttle` method is defined in apache-camel-1.6.4/components/camel-scala/src/main/scala/org/apache/camel/scala/dsl/SAbstractType.scala, Line 85).

Related Patterns

CLEAN METHOD CALLS can be combined with all other patterns.

4.3.2 *Custom Return Objects*

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 results in an undesired syntax which is too much tied to language internals.

Problem

How to improve the domain-specific notations that can be used to access data in multiple structured objects? Accessing multiple-structured data through its native expressions impairs the readability of DSL expressions.

Solution

Create a flat object that defines domain-specific methods to access the data.

```
Feature.all.[2]
```

$$\downarrow$$

```
Feature.all.fetch :feature => :Car
```

The method that normally returns the multiple-structured data instead defines and instantiates an anonymous class with suitable functions that filter the data. An instance of that class is returned to the caller.

In Ruby, `Struct` classes can be used. Python uses `dict` objects per object to store available methods: Simply create an instance of the object that is to be returned, and add the domain-specific notations to the `dict`. In Scala, case classes can be used.

Design Goals

– *Notation* – Use domain-specific keywords to access the data.

– *Generalization* – The custom return object can provide a dedicated focus where specialized domain-specific behavior is introduced.

Liabilities

– When anonymous classes and similar constructs are not available, using named classes populates the namespace of the DSL, possibly conflicting with other entities.
– Directly calling methods on a CUSTOM RETURN OBJECT hides the fact that an object is returned, which could mislead users.

Known Uses

– In Ruby, *ActiveRecord* queries, like `clients.find_by_name("Sebastian")`, return single values or arrays of records, which are typically included in other statements too (the definition of the `find_by_*` methods happens dynamically when methods starting with this syntax are called, which occurs in activerecord-2.3.5/lib/active_record/base.rb, Lines 1839–1961).
– In Python, we could not find this pattern in the analyzed DSLs.
– In Scala, the *Apache Camel DSL* example `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-1.6.4/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 in the same directory).

Related Patterns

– METHOD CHAINING – A CUSTOM RETURN OBJECT can be used as a flexible context provider or referrer.

5. DISCUSSION

In this section, we explain how the choice of principles that a DSL should provide leads to a choice of applicable patterns, and we discuss the various relationships between the notation patterns.

5.1 Principle Support by Patterns

The following ►Table 4 shows which principles are supported by which pattern. Developers that wish to produce a DSL with very specific properties can make a selection of the principles that they want to address, and then use this table to quickly find the patterns that help them in achieving this principle.

Table 4 – Principle support by pattern.

Pattern		Abst.	Gene.	Opti.	Nota.	Comp.	Absorp.
Layout	Block Scope	✓			✓	✓	✓
	Method Chaining				✓	✓	
	Keyword Arguments	✓			✓	✓	✓
Expression	Seamless Constructor				✓		✓
	Superscope		✓		✓	✓	
	Entity Alias	✓			✓		
	Operator Expressions		✓		✓	✓	
Support	Clean Method Calls				✓	✓	
	Custom Return Objects		✓		✓		

5.2 Notation Pattern Relationships

The notation patterns have rich set of relationships among each other. They can be combined in their expressions, used as replacement, or complement each other. We show all identified relationships in ►Figure 2 and further detail these relationships in the following paragraphs.

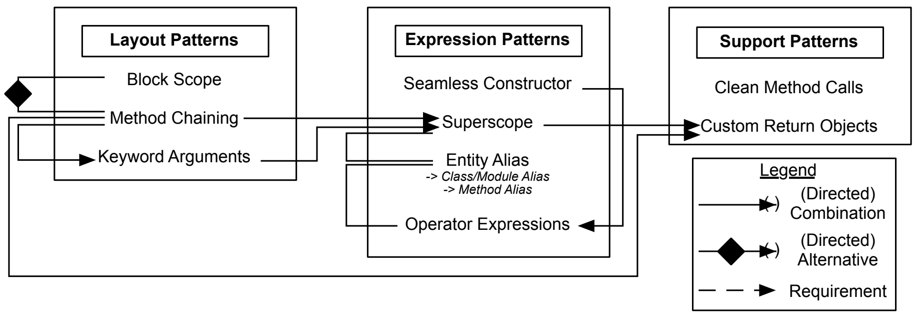

Figure 2: Notation patterns relationships.

5.2.1 *Layout Patterns*

The layout patterns are BLOCK SCOPE, METHOD CHAINING, and KEYWORD ARGUMENTS.

BLOCK SCOPE is a common pattern in DSLs, which is especially often to see in Ruby-based DSL. The positive impact on readability achieved with BLOCK SCOPE is the distinguished, vertical alignment of expressions. Related expressions are grouped inside a block, and the natural hierarchy of objects can be expressed easily.

METHOD CHAINING and KEYWORD ARGUMENTS are both an alternative and an addition to BLOCK SCOPE. These patterns align expressions horizontally, up to the point of looking similar to natural language. METHOD CHAINING is more expensive to implement, because several method declarations have to be made. But KEYWORD ARGUMENTS can be arranged in a similar structure, and they are easy to extend because their declaration occurs in one place only.

The layout patterns can be combined too. In the vertical layout of a BLOCK SCOPE, we can use METHOD CHAINING and KEYWORD EXPRESSIONS for complex expressions. The syntactic relationships between the object that opens the block and the expressions in the block can be very declarative of the particular domain concern.

5.2.2 *Expression Patterns*

The expression patterns are SEAMLESS CONSTRUCTOR, SUPERSCOPE, ENTITY ALIAS, and OPERATOR EXPRESSIONS.

The SEAMLESS CONSTRUCTOR removes the need to use the `new` operator for creating 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.

SUPERSCOPE is a versatile pattern that enhances the flexibility of DSL expressions. It is essential to refer to objects in the expressions, but not always should or can these objects exist in the same scope of the expression. By using a string or similar object, and resolving this reference to the concrete entity, solves this problem.

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. Depending on the used host language, ALIAS ENTITY can be used to customize the complete language to give it a more domain-specific notation.

The final pattern here is OPERATOR EXPRESSIONS. In some domains, it is very intuitive to add elements with a "+", to remove elements with "-", or to check the existence of a particular property with "?". These operators can be defined as simple method calls on chosen objects of the language, enabling convenient expressions.

There are some interesting combinations here. For example, we can provide OPERATOR EXPRESSIONS as an ALIAS ENTITY for built-in methods, like to add modules via the plus symbol. We can even use such a symbol to express that new instances can be created, which means to provide a SEAMLESS CONSTRUCTOR in the form of a symbol.

5.2.3 *Support Patterns*

Support patterns are CLEAN METHOD CALLS and CUSTOM RETURN OBJECTS.

When it is supported, CLEAN METHOD CALLS are used extensively. Although its intent is simple and its application trivial, it has a strong effect on the expressions. Like we showed in the solution section of this pattern, to leave out parentheses, or to use alternative forms for non domain-specific symbols, helps to clarify an expression's intent.

Finally, CUSTOM RETURN OBJECTS is a support pattern for BLOCK SCOPE, METHOD CHAINING, and KEYWORD ARGUMENTS. It replaces complex array access by a more convenient object and property notation. CUSTOM RETURN OBJECTS is especially helpful in a method chain, where it enables multiple dispatch of arguments, and therefore allows to remove some of the extra methods that would otherwise be required to be implemented.

6. SUMMARY AND OUTLOOK

DSLs provide helpful abstractions and notations that allow their users to express domain concerns clearly. The particular challenge with DSLs is their development: Although the host-language characteristics ultimately form the syntax and semantics of the DSL, studies of 12 DSLs in the three programming languages Ruby, Python, and Scala as well as experiments by the author shows that there are recurring challenges and solutions in DSL design. This paper's contribution is a detailed explanation of essential notation patterns that form the DSL's syntax. In total, nine patterns were explained that are concerned with giving the DSLs its general layout, improve individual expressions by providing more concise expressions, or generally by removing parentheses and other tokens from the expressions that have no meaning in the DSL. Additionally, we explained six design principles that provide the essential properties of the DSL. We showed how the design principles are applied by identifying particular design goals for each pattern: a design goal determines for what purpose a pattern can be used, and this helps to shape the DSL in accordance with the principle.

In addition to these findings, our other research also documents the semantic patterns that implement the DSL abstractions and a development process that further structure the application of the patterns [22].

Future research is concerned to extend these research results in three directions. First, although the patterns were found in 12 DSLs, extending the study to other DSLs may uncover new design patterns that need to be added to this catalog. Second, it is interesting to think about how a programming language's basic paradigm shapes the availability of the DSL design patterns. Ruby, Python, and Scala are all languages with a strong object-oriented core and some support for functional programming. Choosing languages that are entirely based on functional programming, such as List or Clojure, or languages that have a rich metaobject protocol, for example Smalltalk, influences the availability of the found pattern. And of course, DSLs in such languages may also use patterns that we did not yet discover. Third and finally, there are research-oriented programming languages that allow to extend the syntax and semantics: Kathadin [31] uses a powerful metaobject-protocol for the extension, and Nemerle [32] uses macros. Using such languages as the host for internal DSLs has a great impact, and the added freedom for syntactic and semantic design could lead to even more concise DSLs.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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