Nr.: FIN-016-2009

Enabling Feature-Oriented Programming in Ruby

Sebastian Günther, Sagar Sunkle

Very Large Business Applications Lab

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-016-2009

Enabling Feature-Oriented Programming in Ruby

Sebastian Günther, Sagar Sunkle

Very Large Business Applications Lab

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

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

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

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

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 04.12.2009

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

Enabling Feature-Oriented Programming in Ruby

Sebastian Günther, Sagar Sunkle

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

Abstract. Feature-oriented programming (FOP), captures requirements and functionality of software at a higher level of abstraction. Modular software can be built using features which are both conceptual entities in the analysis phase and concrete entities in the design and implementation phases of software development. Most research in FOP addresses static languages like Java and C++. In this paper, we describe FOP in the context of the dynamic programming language Ruby. Ruby is a dynamic language featuring a fully object-oriented implementation and rich metaprogramming facilities. Developers can manipulate the program behavior to a large extent, which gives flexibility in designing software. In order to implement features, three programming mechanisms can be identified: Basic expressions (object-oriented mechanisms), metaprogramming and reflection (runtime modifications of variables and methods), and holistic manipulations (treating the program as a string). We discuss and compare these mechanisms for their ability to satisfy the four central feature properties namely, *naming, identification, expressing,* and *composition.* Furthermore, we sketch two implementations of FOP in Ruby. The first implementation uses annotations in the form of comments to express feature-related code. The second implementation one is a DSL which adds features as types to the Ruby language.

1 Introduction

Software development is a complex task. In today's applications, development dimensions like multiple requirements, domains, technologies, and paradigms must be combined. If those dimensions are tightly coupled, several software defects can emerge, such as tangled and bloated code [11], complex function call hierarchy, and a cluttered design. One of the many solutions to this problem is feature-oriented programming (FOP) [18].

Features are characteristics of software used to distinguish members of a program family [2]. *Concerns* describe the general requirements of the software's stakeholder [9] or advances in functionality [10]. Features can be seen from two viewpoints. In the analysis phase of a software, *conceptual features* represent a concern that the software incorporates. In design and implementation of the software, the conceptual features are realized as *concrete features*. A concrete feature is the sum of all changes which need to be applied to the base program in order to achieve new functionality. Different approaches exist, like mixin-layers [19], AHEAD [2], and aspectual feature modules [1].

Current FOP implementation approaches have a number of problems as documented in [16, 10, 21]. Our motivation is to design a new implementation which helps to remedy these problems. As most of the research in feature implementation uses static languages like Java and C++, a natural step is to take a look into dynamic languages. The goal of this report is to structure, explain, discuss, and evaluate how FOP can be achieved in a dynamic language. The target language is Ruby, which is chosen for its fully object-oriented character, its dynamic nature, and the rich metaprogramming facilities it offers.

Our understanding of FOP is that of a *paradigm* realized with an *implementation* which adds types for features to a target language. In this way, features become first-class entities of a host langauge [20, 21]. The first step is to define the properties that such an implementation should possess. We define following properties as derived from [15]:

– ***Naming*** The conceptual features are means to abstract the core functionality of a program. In the analysis phase of software development, these conceptual features are given a unique name and intent. Continuing with the specification and design, the name of the concrete feature should be the same as of its conceptual feature, while the intent is expressed with the *identification* and *composition* properties.
– ***Identification*** The identification property is understood as the task to identify the parts of a program which belong to a feature. Two kinds of granularity have to be considered [10]. *Coarse-grained features* can be thought of as stand-alone parts of the program - they cleanly integrate with the base program via defined interfaces or by adding new classes. On the contrary, *fine-grained features* impact various parts of the source code: extending code lines around existing blocks of code, changing parameters of methods, or adding single variables at arbitrary places.

– ***Expressing*** The identified parts now have to be expressed. In principal, feature related code can be expressed in an external form to the program, like configuration files, or internally in the program using extended custom syntax.
– ***Composition*** Finally, composition is the task which combines the base program with the feature-related code to a coherent and valid form. Of interest are the principal composition mechanism used in a given feature implementation technique and the order in which features are composed.

These properties are central to the proposed implementation approach. Thus, a feature implementations need to name, identify and express features and the code-parts they have in a program, followed by a flexible composition mechanism which combines a base program with the features for a valid and compilable/interpretable variant.

In order to build such an implementation, we spilt our research into two parts. The first one explains the principal *composition mechanisms* which developers can use with Ruby. We will see how Ruby's *basic expression*, *reflection and metaprogramming*, and *holistic manipulations* helps with the implementation. Building upon this knowledge, the next part explains two basic implementations: one with annotations, and one as a Domain-Specific Language. Each part will discuss in detail how the mentioned properties are supported.

2 Ruby Basics

The Ruby programming language was invented in the year 1994. Originating in the context of Unix, it was used as a scripting language for system programming. Libraries for HTTP, FTP, database and graphical user interfaces were added over time. The most important boost to Ruby's popularity came with the introduction of the Rails framework in the year 2004. Rails is one of the most modern frameworks for designing web applications. The easy syntax of Ruby, combined with a set of useful conventions, led to applications that were extremely fast to develop and easy to maintain. Today, Ruby and Rails are becoming serious alternatives to older languages like Java, C++ and the .NET framework.

This section presents Ruby's class model, object creation, language internals, and program decomposition. It is not the task of this report to fully introduce the Ruby programming language, but to explain the most important concepts directly used in achieving FOP. Both [5] and [22] are used as references for the next sections.

Starting from here, we will uniformly assign the following formatting: *keywords*, `Ruby expressions and objects` and FEATURES.

2.1 Core Objects

The most important objects in Ruby which define the overall semantics of any program are *Proc*, *Method*, *Class* and *Module*:

- **Proc** Procs are very versatile objects in Ruby. They can be referenced by a name or created anonymously e.g. in the context of method invocations as an additional argument. When defined, Procs can either reference variables of their surrounding scope, working as closures, or reference variables which yet have to be defined. Further more, they can receive arbitrary number of arguments.
- **Method** Methods consist of the methods name, a set of (optional) parameters (which can have default values), and a body. Methods are defined inside modules and classes. A method is either a *instance method* of a instance object, or a class method of a `Class` or `Module`. Inside classes, methods can have a different visibility, like *private* or *protected*.
- **Class** A Class defined with the notation `class Name; ...; end`. A class consists of its name and its body. Each class definition creates an object of class `Class`. New objects of classes, called instances are created with the `new` keyword. Classes can have different relationships. First, they can form a hierarchy of related classes via single sub-classing, inheriting all methods of their parents. Second, they can mix-in arbitrary other modules, leading to a "multi sub-classing".
- **Module** A Module is defined with the notation `module Name; ...; end`. Modules have the same structure as classes. Class methods in modules can be called on the module, but its instance method can only be used if a class mixes-in the module. Modules can't have subclassing-like inheritance relationships.

2.2 Class Model

The fact that everything inside the Ruby language is a true object is best explained with the class model. An example is shown in ►Figure 1.

We first explain the relationships inside this diagram:

- **Inheritance** Objects can inherit from other classes, gaining access to their methods. Parent classes are called superclasses in Ruby.
- **Singleton** Each objects has a so-called singleton class. The singleton class holds the instance and class methods for their corresponding object. Each singleton class is private to its object. Singleton classes are used fairly often to customize specific objects for a specific purpose.
- **Mixin** Relationship between modules providing their methods to other objects.
- **Instance** are basic in-memory constructs that just contain a value and a pointer to the class of which they are an object off. Although they appear as having all properties and methods defined, their real implementation is the class they belong to.

Let's explain further details about ►Figure 1. The `String` object called *hello* is an instance which just contains the value *"hello"* of the string, and a pointer to its class. The class `String` defines the typical methods to create and modify

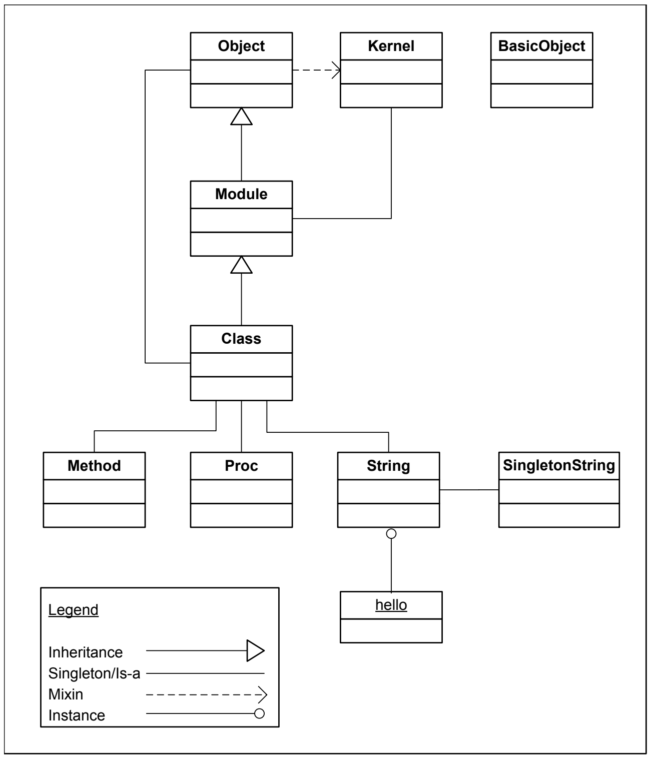

Fig. 1: Ruby's class model

string objects. Whenever a method is called on an instance object, the methods of the class are actually executed. By providing the `SingletonString`, we can change the basic behavior of all string objects. For example, we could store a timestamp-variable with each instance to record its creation time. Going up the ladder, `String` itself is a `Class` object. `Class` contains the essential `new` and `initialize` methods which are needed to create new instance objects. That is to say, when the string *"hello"* is created, the `initialize` method of the `Class` object named String is executed.

At the class models top are `Object` and `Module`. `Module` defines the majority of methods concerning properties of methods themselves (like visibility), and a number of metaprogramming methods (like hooks for adding methods to objects). `Object` is the superclass of all Ruby objects, and at the same time a `Class` object itself[1]. It includes methods for freezing and copying any objects. These methods however are not part of `Object`, but belong to the `Kernel` module, which

[1] This explains the Ruby "Koan" `Class.is_a?(Object)&& Object.is_a?(Class)`.

is mixed-into `Object`. Finally, the new Ruby version 1.9 added `BasicObject` as an ancestor for objects which only need a minimal amount of methods.

The refined terminology helps in understanding what the core of Ruby is - flexible objects with a details hierarchy, which can be extended and modified at runtime.

2.3 Type System

Ruby does not have a static type hierarchy for its classes and objects. Ruby objects are identified according to the methods they provide. An object of any class can be used in all contexts as long as it responds to the required method calls. In the Ruby community, this is called "duck typing": "If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck"[22].

Although this may seem unorthodox to the developers used to working with typed languages, there are a number of benefits. First of all, the amount of code is reduced. Second, what kind of object is actually used in an expression can be changed at runtime as long as the new object responds to the required methods. And third, new objects can be added anytime to the program. On the downside, this requires some programming discipline by using consistent objects within the program, but the advantages exceed the disadvantages.

3 Graph Product Line

Since explaining a programming language without programs is futile, we introduce the Graph Product Line as the running example for all following code listings. The Graph Product Line (GPL) describes a family of related application in the domain of graphs. It is often used in the context of FOP as a case study for implementation [14]. We see a graphical representation of the features, their relationships in ►Figure 2.

The root node is GPL, representing the graph product line itself. It is followed by four mandatory features: TYPE, WEIGHT, ALGORITHMS and SEARCH. The TYPE specifies whether the graph is DIRECTED or UNDIRECTED, and WEIGHT whether the graph is WEIGHTED or UNWEIGHTED. These properties are exclusive: A graph can't be DIRECTED and UNDIRECTED at the same time. The next feature is ALGORITHMS: NUMBER traverses the tree and assigns unique numbers to the nodes, CYCLE calculates if cycles are included inside the graph, and SHORTEST computes the shortest path from a given source node to others. All these features are optional. And finally, the SEARCH is either BFS (breadth-first search), DFS (depth-first search), or NONE.

A efficient implementation of the graph algorithms would use matrices. While this is the implementation to choose for building libraries, in the context of a study about FOP, it was suggested to use the natural domain entities in the program [14]. This means to implement classes like *Node*, *Edge* and *Graph* as classes.

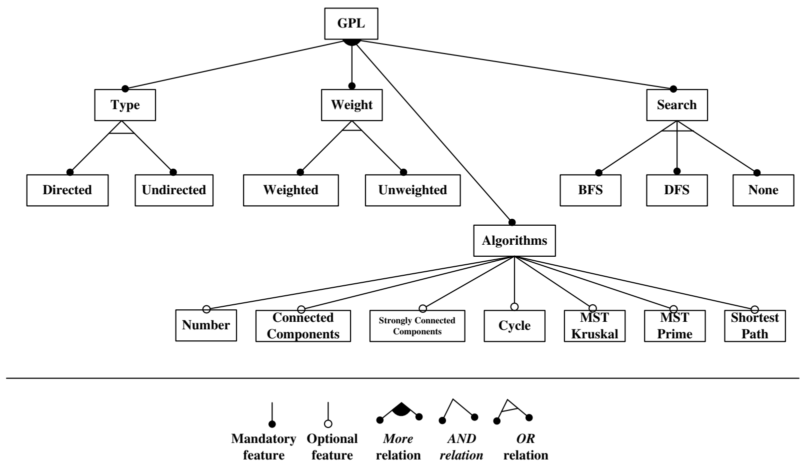

Fig. 2: The Graph Product Line

4 Programming Mechanisms

Ruby has a rich set of overlapping programming mechanisms. When we studied them, the various mechanisms formed a natural order according to their power to reference and change parts of a program. In total, three mechanisms were identified. The *basic expressions* are mechanisms stemming from the object-oriented paradigm and Ruby's dynamic nature. At the next level, *reflection and metaprogramming* allow finer ways to select parts of the program and modify it. Finally, the *holistic manipulations* allow selecting and modifying arbitrary pieces of code by treating the whole program as a string. Each mechanism is presented thoroughly in the following subsections. The focus thereby is how to select and manipulate the aforementioned building blocks of a Ruby program: Procs, Methods, Classes, and Modules.

4.1 Basic Expressions

This section discusses four basic mechanisms helping in archiving the feature properties: *Method objects*, *subclassing*, *mix-in*, *reopening*.

4.1.1 Method Objects `Proc` is one of the most versatile objects in Ruby. Procs allows storing arbitrary code and executing it in another scope. Methods are closely related to Procs - the only difference is that methods have a defined place where they are valid. While `Method` objects can be called immediately, `UnboundMethod` need a so-called binding object which is of the same class as the object the original method was defined in. The interesting feature of any `Method` object is that it can be handled and called like a `Proc` object.

We show an example for working with `Method` objects. In ►Figure 3, line 1, the programmer stores the method to `Node#set_weight`[2]. We can see that the result is a `Method` object in line 2. Finally, we use `call` in line 3 to execute the method and pass an argument. The same syntax would also apply to `Procs`.

```
1 meth_body = Node.method :set_weight
2 meth_body.inspect
3 => "#<Method: Node#set_weight>"
4 meth_body.call 3
5 => "3"
```

Fig. 3: Creating and using a `Method` object

Some example use-cases for methods are to export functionality to another object and to call the method in a free context.

4.1.2 Subclassing A classic method of object-oriented programming is to define a parent class which includes core methods, and let child classes inherit the same functionality. The subclasses can add or overwrite methods for specific purposes. In ►Figure 4 we see how to first define a `BasicNode` class (line line 1 to 8) which is then subclassed by `NeighborNode` in line 10. Line 14, `attr_reader`, is a meta-method to define read access on instance variables.

```
 1 class BasicNode
 2   def initialize source, sink, weight, directed
 3     @source = source
 4     @sink = sink
 5     @weight = weight
 6     @directed = directed
 7   end
 8 end
 9
10 class NeighborNode < BasicNode
11   def add_neighbor(node)
12     @neighbor << node
13   end
14   attr_reader :neighbor
15 end
```

Fig. 4: Using subclassing to combine the code of a `BasicNode` with `NeighborNode`

Subclasses preserve the scope of constants, variables and method visibility for their children. They are a natural vehicle for sharing functionality. Furthermore,

[2] We apply the following notation: `Node#set_weight` is a instance method, and `Node.set_weight` is a class method.

the subclassing mechanism can be used to mirror the structure of the domain they reflect.

4.1.3 Mix-In The third mechanism in the basic expressions is to use modules and the mix-in functionality. Modules can define inaccessible instance methods using the normal `def name ... end` notation. These instance methods can be called by either modules or classes using two mix-in operators. With `extend`, a module adds its instance methods as class methods to the calling module or class. And `include`, called on a class, makes the modules instance method as the classes' instance methods available.

In ►Figure 5, we define the `Neighbor` module (line 1) which contains the `add_method` (line 2) and `attr_reader` (line 5) as before. By using the `include` statement in line 9, `Node` now contains `Neighbor's` methods.

```
 1 module Neighbor
 2   def add_neighbor(node)
 3     @neighbor ||= []
 4     @neighbor << node
 5   end
 6   attr_reader :neighbor
 7 end
 8
 9 class Node
10   include Neighbor
11 end
```

Fig. 5: Using a mixed-in module to compose the `Node` class

Modules provide a clean, namespace secured mechanism to share functionality and constants with other modules and classes. Their independent definitions makes them the premier vehicle when other classes and objects needs to include several methods of different origin. As explained in the class model of section 2.2, objects mixing-in a module merely contain a pointer to that module. So it is easy to update functionality in a central module, and have it ready in other objects too.

4.1.4 Reopening Modules and classes are open objects. Their definitions can be opened again anywhere in the course of a program. Typically, new methods are introduced or existing ones are overwritten[3].

In the following ►Figure 6 we first see how the class `Edge` is defined to have a basic `initialization` method (line 2). When implementing some algorithms in the GPL, we see the need to store neighboring nodes as well. So, later in the

[3] Ruby does not directly provide overloaded methods. However, using a hash with named parameter, arbitrary length arguments are often realized. This is called *named parameters*.

code, we open the definition again and add the `add_neighbor` method (line 11) and a `attr_reader` (line 14) for the new neighbor instance variable. The combined `Edge` class has a `initialize` method and the `add_neighbor` methods, plus the five instance variables `@source`, `@sink`, `@weight`, `@directed` and `@neighbor`.

```
 1 class Edge
 2   def initialize source, sink, weight, directed
 3     @source = source
 4     @sink = sink
 5     @weight = weight
 6     @directed = directed
 7   end
 8 end
 9 ...
10 class Edge
11   def add_neighbor(node)
12     @neighbor << node
13   end
14   attr_reader :neighbor
15 end
```

Fig. 6: Defining class `Edge` first with its initialize method, and later extending it with a `add_neighbor` method

Ad-hoc reopening of objects to change their definition should be used carefully. The alternatives, subclassing and mix-in, provide a more consistent approach for expressing changes. Reopening is however often the only non-metaprogramming option to access objects' otherwise hidden information. As any code is executed in the context of the object, it is possible to read e.g. internal instance variables not accessible from the outside.

This completes the explanation of the basic expressions. We continue with the explanation of the reflection and metaprogramming mechanisms of Ruby.

4.2 Reflection and Metaprogramming

Reflection and metaprogramming play a vital role in Ruby. Because they are built into the language and are convenient to use, many Ruby programs, e.g. open source applications, use these facilities. We draw a line between the two closely related concepts: In Ruby, reflection methods return values naming Ruby objects, and metaprogramming uses these identifiers to modify the objects. We explain them separately. The material discussed here again stems from [5] and [22]

Reflection is the ability to derive internal information about a program. In Ruby, reflection methods allow convenient access to (i) methods for objects grouped by their visibility (public, private, protected) and their type (class, instance), (ii) global and instance variables, and (iii) constants. Table 1 shows how Ruby's core objects become addressable through the reflection methods.

Table 1: Ruby's reflection methods

Object	**Object Property**	**Applicable methods**
Module/Class	*Name*	ObjectSpace#each_object, Class#superclass, Module#class, Module#nesting, Module#ancestors, Module#included_modules
	Methods	Object#protected_methods, Object#public_methods, Object#private_methods, Module#public_instance_methods, Module#private_instance_methods, Module#protected_instance_methods
	Class variables	Module#class_variables
	Instance variables	Object#instance_variables
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-
Method	*Name*	(via Module/Class methods)
	Parameters	-
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-
Procs	*Name*	-
	Parameters	-
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-

We see that the modules, classes and methods can be addressed, but the body internals are not visible. Only one method is applicable to derive some information about the body - `Kernel#local_variables` - but this method is also constrained because it can only be called inside the body and returns just names of the variables, no pointers. Because the methods are straight-forward to use, we restrain from giving a detailed explanation, and continue with explaining the metaprogramming methods of Ruby.

Defining metaprogramming according to [23], Metaprogramming is the concept that program synthesis is a computation." Ruby provides sophisticated metaprogramming methods, which allow the following:

– Class and instance variables can be added and removed, as well as constants
– Methods can be added (instance and class method), copied or deleted
– Arbitrary code, executed in the context of its receiver, can be used to define inner classes, change the visibility of methods and more (the various `eval` methods).

Table 2 lists in detail the available methods and short descriptions.

We want to give an example how to use the metaprogramming methods for extending functionality of a class. In ►Figure 6, we explained how to extend the `Edge` class with methods for setting the neighbor. We used the reopening mechanism back then, and now show how to do the modification with metaprogramming.

In ►Figure 7, we first define the method to be extended as a string (line 1 - 5) and as a proc (line 7-12). The actual extension is a simple method call: - `class_eval`, called in line 17 and 18 with the string respective the proc.

Table 2: Built-In metaprogramming methods

Object	**Method**	**Explanation**
Variables	Module#const_set, Module#remove_const	Set and remove constants
	Module#attr, Module#attr_reader, Module#attr_writer, Module#attr_accessor	Provide read, write or combined access to instance variables
	Object#instance_variable_set, Object#remove_instance_variable	Set and remove instance variables
	Module#class_variable_set, Module#remove_class_variable	Set and remove class variables
Methods	Module#alias_method	Copies method body to a newly named method
	Module#define_method	Defines new methods
	Module#undef_method	Prevents object to respond to calls of the method
	Module#remove_method	Deletes the method from module/object
Arbitrary	BasicObject#instance_eval	Evaluates string or block (defines class methods)
	BasicObject#instance_exec	Evaluates block with additional parameters (defines class methods)
	Module#module_eval, Module#class_eval	Evaluates string or block (defines instance methods)

```ruby
neighbor_string = "
def add_neighbor(node)
  @neighbor << node
end
attr_reader :neighbor"

neighbor_proc = lambda do
 def add_neighbor(node)
   @neighbor << node
 end
 attr_reader :neighbor
end

class Node < BasicNode
end

Node.class_eval neighbor_string
Node.class_eval &neighbor_proc
```

Fig. 7: Extending class `Node` with string or proc object to add the `add_neighbor` method

Comparing reopening and metaprogramming shows that each provides similar methods. At first, combining reflection and metaprogramming allows to use runtime knowledge about dynamic created objects which become subject to modifications. With reopening or other basic expressions, the objects to be modified must be known in advance. Second, reflection and metaprogramming use defined keywords to clearly express their intent. Runtime changes are more concise to express with this. In summary, developers should use basic expression for pre-runtime modification, and metaprogramming for runtime modification.

But like the basic expressions, reflection and metaprogramming have one weakness: Once evaluated, the body of modules, classes, methods and procs become hard to extend - adding and overwriting are the only alternatives. Since immutable bodies prohibit fine-grained features, other methods are still needed. Here the holistic manipulations enter the field.

4.3 Holistic Manipulations

The internals of Ruby's core objects can not be modified with basic expressions or reflection and metaprogramming. This prevents implementing fine-granular changes to the program. But there are two mechanisms that enable this: *string manipulation* and *abstract syntax tree manipulation*.

4.3.1 String Manipulation Ruby is basically implemented as text. Techniques to identify patterns inside a text allow both coarse-grained and fine-grained identification and modification of source code. A natural way of string manipulation is to use regular expressions.

The Ruby class Regexp allows sophisticated and convenient methods for working with regular expressions [22]. A regular expression, or "regexp" in the Ruby jargon, is created with a specialized syntax. Patterns of strings and numbers form the background. Further options are patterns with character classes (strings, numbers, whitespace), anchor (beginning or end of line), repetitions of values, and the use of lookahead or lookbehind. With these patterns, selections for (i) the name of the method, (ii) properties or appearance of the methods parameters, (iii) properties of the methods body (variables, length, and specific line), (iv) position of the method, and many more can be used. Each time a match occurs, special variables for the exact match, the text before, and the text after the match, are created. The match itself can be sub-grouped where each group is available through the `MatchData` object.

In ►Figure 8, we open the file "source.rb" and iterate over all text lines. For each line, we check whether the line contains a method declaration at the beginning (line 2). The regexp is defined in backslashes (\ ..\). The first part is the `def` keyword which we are looking for. Followed by a space, the expression `\w+` states to select text characters which have at least one occurrence - which is the methods name. If a match occurs, the global variable `$1` contains the methods name.

```
1 File.read("source.rb").each_line do |line|
2   if line =~ /def (\w+)/
3     ...
4   end
5 end
```

Fig. 8: Regular Expression for matching method definitions

Regular expressions can be used to process the static structure of a Ruby program - the source code prior to execution. Changes at runtime are expressible if parts of the source code are evaluated again. Runtime modifications which are not defined in static files, but e.g. happen through user interaction, can be tracked with some effort. Here, the library *Ruby2Ruby* comes into play. At runtime, arbitrary expressions, including modules and classes, are transformable to a string representation. Consider ►Figure 9 where we first transform the `neighbor_proc` (line 1) of ►Figure 7 and then the whole class definition for `Node` (line 3).

```
1 Ruby2Ruby.translate neighbor_proc
2 >>  "def add_neighbor(node)\n  (@neighbor << node)\nend"
3 Ruby2Ruby.translate Node
4 >>  "class Node < BasicNode\n  def add_neighbor(node)\n
      (@neighbor << node)\n  end\n  \n  attr_reader
      :neighbor\nend"
```

Fig. 9: Transforming a proc and the complete class definition of `Node` back to a string

Combining regular expressions with Ruby2Ruby allows for string manipulations which target coarse-grained and fine-grained feature definitions even at runtime.

4.3.2 Abstract Syntax Tree Manipulation In general, computer programs are handled to a translator (compiler or interpreter). The translator scans the input and creates a token list form it. Next step is to transform the token list to an Abstract Syntax Tree (AST). Abstracting from the concrete source code, the AST of a program captures the name, precise scope and semantics of any expressions. Various checks are done to the AST, optimizations, and finally compilation or interpretation. The AST is the immediate representation of a program which is used for further processing and code execution. Manipulations of an AST mean the manipulation of the program itself.

Ruby provides runtime modification ability for its own AST. With the help of the library *ParseTree*, a nested array representing the AST can be shown for any expression or in-memory objects accessible by reflection methods (like classes and methods).

►Figure 10 shows the parse tree for the method definition of `set_weight`. In line 2, the `defn` specifies a function definition whose name is followed in line 3. The nested array starting at line 4 is rather complex. It defines a block of code which receives a argument called `weight`. Then it is followed by an assignment `iasgn` of the variable `@weight` to the value `weight`.

For understanding how modification to a program are done using AST manipulations, we resort to the mix-in example in 5. This time, the class `Node` inherits from `BasicNode` which includes common node functionality.

►Figure 11 shows the parse tree for the class definition. Line 1 and 2 state that the expression is a class with the name `Node`. The subclass relationship is

```
ParseTree.translate "def set_weight(weight); @weight = weight; end"
>>  [:defn,
   :set_weight,
   [:scope,
     [:block,
       [:args, :weight], [:iasgn, :@weight, [:lvar, :weight]]
  ]]]
```

Fig. 10: The parse tree for the `set_weight` method

defined on Line 3: Node has a reference to the constant BasicNode. On line 4 and line 10, two methods are defined. The first is `add_neighbor`. It literally receives a `block` with the arguments `node`, and then calls the method `<<` with `neighbor` as the left value and `node` as right value. On line 10, the `attr_reader` is translated to the function call `neighbor` which returns the instance variable `neighbor`.

```
ParseTree.translate Node
>> [:class,
   :Node,
   [:const, :BasicNode],
   [:defn,
    :add_neighbor,
    [:scope,
     [:block,
      [:args, :node],
      [:call, [:ivar, :@neighbor], :<<, [:array, [:lvar, :node]]]]]],
   [:defn, :neighbor, [:ivar, :@neighbor]]]
```

Fig. 11: Parse tree for the Node class

In ►Figure 12, using the AST, we change the superclass of `Node` to the `AdvancedNode` - this class contains a setter for the weight. We further add a reader for the nodes weight directly in `Node`. In line 2, we select the appropriate array position to change the superclass to the value `:AdvancedNode`. Then, we add the method definition after the defined `attr_reader` for neighbor (line 4). Line 6 shows the back-translated string representing the modified node. To actually apply this change, we need to call the `class_eval` metaprogramming method introduced in section 4.2.

With holistic manipulations, we are finally able to access also fine-grained modifications. The example showed the prototype for selecting and updating the AST representation. Using this method in production code however would require making the changes more convenient. We could encapsulate the method calls, and build convenient methods which would apply semantic changes like "update the superclass" or "add a new line to the method body".

```
tree = ParseTree.translate Node
tree[2][1] = :AdvancedNode
tree[5] << [:defn, :weight, [:ivar, :@weight]]
Ruby2Ruby.translate_tree tree
>> "class Node < AdvancedNode\n  def add_neighbor(node)\n
     (@neighbor << node)\n  end\n  \n  attr_reader :neighbor\n  \n
       attr_reader :weight\nend"
```

Fig. 12: Manipulating the AST and convert it back to a class definition

4.4 Evaluating the Feature Properties

Having explained the available mechanisms for code manipulation in Ruby programs, we will now discuss how the mechanisms are applicable to develop an implementation of FOP. The next sections will successively explain how the *naming, identification, expressing*, and *composition* properties of features are achievable in Ruby. Each section will describe the general nature of the property and then discuss the benefits and disadvantages of the mechanisms.

4.4.1 Naming Each feature needs to have a unique name which identifies it. Ruby provides the two entities *variables* and *constants* [22].

- **Variables** are mutable objects with different scoping. Global variables are seen from any scope. Instance and class variables are visible inside their object respective their class, and local variables are only visible inside their surrounding definition scope, like in the body of methods or procs.
- **Constants** are less mutable[4] objects with namespace scoping. Constants defined within modules and classes are accessible unadorned within their modules and (sub)classes. From the outside, they can be referenced by using the name of the module and class plus the double colon `::`. Constants defined in the top level object are viewable without restrictions from each scope. All modules, classes and their nesting furthermore declare constants as well.

Because of the scope, constants should be favored over variables. In principle, every object of Ruby can provide the *naming* property. But what is the criteria guiding this choice? We need to think about the further properties features will have. Features may have a kind of status, such as they are activated or not activated for a particular variant, and they may have behavior themselves. Both modules and classes provide the needed mechanisms. Furthermore, they also allow nesting definitions in each other to represent the hierarchy of a product line. However, only after all other feature properties have been explained we can assess which entities can be used for naming.

[4] The object referenced by a constant can be changed as well as the referenced object is changeable - but not within a method

4.4.2 Identification After the features have been named, the next step is the *identification* of the relevant part of the source code which belongs to a certain feature. Following Table 3 shows how the programming mechanisms are viable to identify parts of the program. We clearly see that only holistic manipulations provide the means for fine-grained modifications of the source code. But do we always need such fine-granular changes? If not, the built-in reflection and metaprogramming facilities provide enough expressive power for the purpose of identification.

Table 3: Applicability of identification methods to identify objects

Object	**Object Property**	**Basic Expressions**	**Reflection and Metaprogramming**	**Holistic Manipulations**
Module/Class	*Name*	✓	✓	✓
	Methods	✓	✓	✓
	Class variables	✓	✓	✓
	Instance variables	✓	✓	✓
	Body (Variable/Procs)	-	-	✓
	Body (Other)	-	-	✓
Method	*Name*	✓	✓	✓
	Parameters	-	✓	✓
	Body (Variables/Procs)	-	✓	✓
	Body (Other)	-	-	✓
Proc	*Name*	-	✓	✓
	Parameters	-	✓	✓
	Body (Variables/Procs)	-	-	✓
	Body (Other)	-	-	✓

4.4.3 Expressing We know that basic expressions only have very limited support to actually extend source code. If we developed an application in which features are expressed entirely in terms of basic expressions, it would require us to manually update and synchronize two infrastructures: The basic program and the features. Feature combinations are problematic too: If they crosscut each other, developers need to write the feature combination manually. This leads to high maintenance effort for complex applications.

Reflection and Metaprogramming are more powerful. It is possible to add or remove individual methods, which is more granular then the basic expression. Modifying methods internals however is not possible with metaprogramming either.

Finally, holistic manipulations also allow changing body internals. In order to function properly however, holistic manipulations need to access the mechanism of basic expressions and metaprogramming to write the changes back into the program. With the string manipulation, we extract the base program, apply the changes, and reevaluate the relevant part so that even runtime modifications are possible. We will see further evidence that only a combination of all programming mechanisms can truly satisfy the properties to enable FOP in Ruby.

4.4.4 Composition Composition is the final process in which a base program and the feature expressions are combined to a valid program. Although the existing mechanisms seem to offer disjoint functionality, they are invaluable complements that can address fine-grained composition only in combination.

Table 4 presents what kind of modifications can be applied by a composition approach. The modifications are *add* (add expressions), *delete* (delete expressions), *overwrite* (replace expressions with other expression), and *extend* (arbitrary expression modification). Usually, *extend* includes *add* and *delete* for whole expressions too. In the table, + means that the modification is added and = means that the modification type remains.

Table 4: Applicability of composition methods to modify objects

Object	**Object Property**	**Object-Oriented Mechanisms**	**Metaprogramming**	**Holistic Manipulations**
Module/Class	*Name*	Add	=	+Extend
	Methods	Add, Overwrite	+Delete	+Extend
	Class variables	Add, Overwrite	+Delete	+Extend
	Instance variables	Add, Overwrite	+Delete	+Extend
	Body (Variables/Procs)	-	-	+Extend
	Body (Other)	-	-	+Extend
Methods	*Name*	Add, Overwrite	=	+Extend
	Parameters	Overwrite	=	+Extend
	Body (Variables/Procs)	Overwrite	=	+Extend
	Body (Other)	Overwrite	=	+Extend
Procs	*Name*	-	-	+Extend
	Parameters	-	-	+Extend
	Body (Variables/Procs)	-	-	+Extend
	Body (Other)	-	-	+Extend

The table shows how important the combination of the mechanisms is, which we want to describe further. We first return to the object-oriented mechanisms. How are the options to use reopening, subclassing, or mix-ins to be evaluated? We suggest that classes should be used for clean application design with a hierarchy of objects that reflect a certain structure inside the application domain. Modules can contain code which either performs global functions or which must be included at different objects inside the class hierarchy. Small changes to individual objects could be done with reopening classes and manually write new methods into the definition. But developers should instead use the metaprogramming methods. The built-in metaprogramming operations allow to name and identify semantic changes which are to be applied to source code. A combination of these approaches provides sound mechanisms for a clear design of the base program.

We now enter the stage of program manipulation. Here, the need for the metaprogramming capabilities is clearly seen. With `alias_method`, the body of a method can be copied, and with `remove_method` unneeded methods can be permanently removed. Further, the metaprogramming methods allow finer control over the changes they apply. But neither object-oriented mechanisms nor the metaprogramming methods can change body internals of modules, classes, methods and procs. At this point, the holistic manipulations are needed. Once the body is transformed to a string or a parse tree with the Ruby2Ruby library, we can use all manipulations imaginable. The importance hereby lies in the accurate expression of the semantics for the applied changes. The circle of the composition mechanisms closes with the holistic manipulation. First, they need the built-in classes to manipulate strings for the regular expressions or arrays for the parse tree, and then they use metaprogramming methods for reevaluating the changed code.

4.5 Summarizing Ruby Programming Mechanisms for FOP

We showed and discussed the principal programming mechanisms basic expressions, reflection, metaprogramming, and holistic manipulations. The result is that only the combination of all mechanisms allows targeting both coarse-grained and fine-grained modifications. Each mechanism should be used with respect to its purpose:

– *Basic expressions* for building the base program in a clear and non-repetitive way.
– *Reflection and metaprogramming* to access the running programs structure and modify it.
– *Holistic manipulations* to change the otherwise invisible body of methods.

We now turn our attention to the implementation prototypes for FOP in Ruby.

5 Enabling Feature-Oriented Programming

In addition to the theoretical discussion, we present two small example implementations. The goal is to discuss two principal methods and evaluate what method is more promising into achieving the FOP implementation which satisfies all feature properties.

First, we describe a solution that utilizes static comments in a program to express code changes related to features. The program is read by a parser which transforms comments into code if the feature is activated, and returns a string representing the program. This string representation is finally executed with the explained `eval` methods.

Second, we present an implementation that is based on the idea of domain-specific languages. Such languages represent their domain as a language [24]. In the case of FOP, this allows us not only to define entities for features, but also to express them inside the Ruby language.

5.1 Annotations

We had one goal in mind when implementing FOP solutions: The code belonging to a certain feature should not be declared outside the program in another file, but close to the place it is actually to be used in. A simple solution to express code related to features is to *annotate* it with a special syntax. We decided to use comments for this.

Ruby has two types of comments. The first one are *single line comments* which begin with the hash sign `#` and stop at the end of the line - it is not possible to comment parts of a line only. The other option is a so-called *heredoc*. These are multiline strings whose delimiter can be chosen freely. ►Figure 13 show how to define a multiline string `heredoc` with the delimiter `EOC`.

```ruby
herdoc = <<-EOC
def set_weight(weight)
  @weight = weight
end
EOC
```

Fig. 13: Example for a heredoc

This is the basic idea for an annotation-based approach. We decided to use both normal comments and heredocs, and to include the information to which feature code belongs either as an additional field in the comments or as the heredoc delimiters.

Let's take a look at an example. Our goal is to define the `Edge` class which includes methods and attributes for its weight if the corresponding WEIGHTED feature is activated. In ►Figure 14 we see this definition using annotations. Line 2 shows an in-line comment which names the WEIGHTED feature with the syntax `F!Weighted#`. The first part of the line would declare a normal `attr_reader` for the `source` and `sink` attributes. Invisible to the interpreter is the `weighted` attribute. We also see an example using heredocs. Line 4 defines the same syntax for defining the feature, `F!Weighted`, and then includes from line 5 to 7 a method declaration.

Once the whole program has been annotated in this way, the composition can begin. The central entity is a class called `FeatureApplier`. Internally, it stores the list of activated features, the single-line annotations per feature and the heredocs per feature, the original source code, and the current delta source code. When a new `FeatureApplier` object is created, it receives a string containing the program

```
class Edge
 attr_reader :source, :sink#F!Weighted#, :weight

 <<-"F!Weighted"
  def set_weight(weight)
    @weight = weight
  end
  F!Weighted
end
```

Fig. 14: Using annotations to extend the `attr_reader` and to define the set_weight method

and parses the string. Whenever it detects either a single-line comment or a heredoc, it constructs the resulting source as if the feature would be activated, and stores it in a hash. After that, users simply call `activate_feature` with a symbol argument representing the feature to be activated. The `FeatureApplier` will select all transformed lines matching the activated code, and replace the relevant part of the original, thus creating a new delta. The newly configured program can be executed by using `Module.eval` with the delta.

We will discuss the properties of the annotation approach thoroughly after presenting the Domain-Specific Language approach.

5.2 Domain-Specific Language

Domain-Specific Languages, or short DSL, have a long tradition in computer science (they go back to a paper by LANDIN [12]), and are becoming a mature concept in software development. Some very recent examples are modern telephone support [13], healthcare systems [17], and web applications [26].

The power of a domain specific language lies in its abstraction. By designing the domain with the "natural" entities and relationships, the domain-knowledge is expressed as a language [24]. Using DSL effectively, advantages like increased productivity, efficient code-reuse, and reduction of errors [4] [7] are achieved.

In our case, we generally see a DSL as a programming language, not as a declarative language which only sets parameters. One compelling characterization of a DSL is whether it is realized as internal or external language [4]).

- A *external DSL* is a separate language in which syntax and semantic can be defined freely. This requires language creators to manually define a compiler or interpreter for the language so that actual working code is generated. Unless the code isn't transformed to the same code as the application uses, this approach has limitations.
- The other option is *internal DSL*. Internal DSL use an existing language as the base. This means that infrastructure consisting of compilers, interpreters and even editors for the host language can be used for designing DSL. Developers customize the language by adding keywords which represent the

domain. Furthermore, they can change - with limitations - the syntax and language internals.

Dynamic languages are more suitable as host languages because their whole definition is open to runtime modification. In the following, we sketch an early DSL for FOP in Ruby.

The central part of the DSL is the way how features are represented. Features are realized as modules with the following functions.

– `activate` Activates the feature and makes the features methods callable
– `deactivate` Deactivates the feature by replacing the feature's method's body with an error message
– `code` Receives a Proc object which identifies the source code belonging to the called feature

We now explain step-by-step how to use the DSL. The first part lies outside using the DSL: One needs to have a model of the product line he wishes to implement. For simplicity, we will reuse the Graph Product Line introduced in section 4. Features are created by including the feature module, as shown in the following ►Figure 15. We see that the features Weighted and Unweighted are defined as features.

```
class Weighted
  include Feature
end

class Unweighted
  include Feature
end
```

Fig. 15: Defining features

In the next step, we analyze what parts of the program belong to the identified features. The respective code, be it whole blocks like method definitions or just individual lines, are put inside a block and called with the `code` method of the `Feature` module. For example, the attribute `weight` is only set in the `Node` class if the Weight feature is set. The code may look like in ►Figure 16.

```
class Node
  attr_reader :value
  Weight.code { attr_reader :weight }
end
```

Fig. 16: The `Weight` class containing code related to the Weight feature

Once all the code is implemented this way, the program is ready to be initialized. They put the whole program inside a Ruby Proc object and hand it over to a module called `FeatureResolver`. This module handles the internals of feature activation and deactivation. The details are as follows:

- Each class which includes the `Feature` module registers itself with the `FeatureResolver`
- The `FeatureResolver` stores the class's name in an Array, and stores all methods in an Hash with the key *class_method-name*
- For the case that the Feature becomes deactivated, altered methods are stored in another array: "Deactive" methods just raise an error saying that the Feature is not activated and the method can't be called. The discussed Ruby2Ruby library is used to parse the method, select all method names, and then to use the method names together with the `instance_method` method to create `UnboundMethod` objects which are stored inside the hash
- If a feature gets activated, the corresponding active methods are defined via the `class_eval` method
- If a feature gets deactivated, the corresponding deactivate methods are defined, again via `class_eval`

Once the program is loaded, features can be activated in arbitrary order - the composition order plays no specific role.

We now continue with the explanation of how the suggested feature properties are satisfied by the annotation approach and the DSL approaches.

5.3 Evaluating the Feature Properties

In the following Table 5, we discuss side-by-side how the annotation approach and the DSL approach score with respect to the feature properties.

We can clearly see that using a DSL has much more potential then annotations. In our view, the biggest advantage is that features are entities of the language, which allows other parts of the program to query them for information regarding their status. Comparing to other FOP approaches, it is also not required to use any external structure, not even a separate source file, to view the code belonging to a feature. Developers can focus more on the code because the identification and expression properties are represented the same entities that implement features. With this result, we decide to advance the DSL approach.

6 Related Work

The support for explicit FOP in Ruby is in a very basic state. To the best of our knowledge, no full libraries have been developed. Outside scientific publications, blogs of Ruby developers assume that Ruby's capabilities are naturally so powerful that no explicit support of FOP is necessary [6]. However, from the viewpoint of software product lines and the need to express variations in the software, this view is to question.

Table 5: Comparing Annotations and Domain-Specific Languages with respect to satisfying the feature properties

Property	Annotation	Domain-Specific Language
Naming	No explicit entities are created, features are just described as names inside comments. Features are not known to other entities, which makes interaction with other program constructs difficult.	Each feature is a class which is represented as a constant too. Defined in the top level, every feature is available and known in all expressions.
Identification	Identification is a external process the developer undertakes. He recognizes which part belong to a feature, and is responsible for putting them in annotations using comments.	The developer is solely responsible for the identification too. Once the belonging parts are found, he uses an extended syntax to describe the feature related code. The code is fully accessible by the program. Furthermore, the developer thus stays more closely to the program he is working on.
Expressing	Central entity is the FeatureApplier, which receives the program as a string, receives activation and deactivation calls, and changes the program string representation accordingly.	Central entity is the FeatureResolver , which receives the program as a proc object. Feature activation happens directly with the features. Code belonging to a feature is updated according to features' activation status.
Composition	Currently only changes the string, which needs to be evaluated in a Ruby environment to work.	Directly applies the changes on the source code basis. Feature are activated typically at runtime, but variants can be configured statically from the outside.

Instead of FOP, we found some alternatives in Aspect-Oriented Programming (AOP). Bryant and Feldt build the first AOP library in Ruby, but their work was not continued since 2002 [3]. One recent approach is Gazer [25]. It is a small language to define aspects as methods refinements, which can call code before or after the method body. An example of a sophisticated AOP implementation in Ruby is Aquarium.

In Aquarium[5], aspects are represented as classes. The classes are subject to various parameters for the scoping of the aspect. These parameters express the classes, methods (concrete name, selected by visibility, confirming to a certain pattern) and the type of the aspect (after, before and around the original source code, or for special cases like raising exceptions). The code to evaluate is given as a block. Aquarium saves the changes applied to methods, and it is possible to reverse them too. Aquarium is subject to further development. Its developers have defined the future scope of the library and are continuously developing next releases.

Another approach to modularize software with dynamic languages in general is context-oriented programming (COP). COP defines independent layer in which objects reside. In each layer, they have changed behavior and data. COP allows to define fine-grained message definition to reflect the different states [8]. ContextR[6] is a COP library in Ruby. ContextR defines a language level expression for COP. Contexts are naturally expressed as context classes. Objects of the domain can activate a certain context to change their appearance or state to

[5] http://aquarium.rubyforge.org/
[6] http://contextr.rubyforge.org/

the outwards. Per context, all objects are visible to each other. The mentioned messages rules are also expressed at language level within ContextR.

7 Summary and Future Work

This paper presented a detailed introduction to FOP. We covered core terminology and characteristics of features, provided the graph product line as a cohesive example and explained shortly the most important properties of the Ruby programming language. After that background material, we researched the mechanisms Ruby gives developers to implement FOP. Naturally, it was difficult to separate the mechanisms. Ruby is very flexible and allows multiple ways to express the programmer's intentions. For method extension alone, Ruby allows four different methods. After detailed descriptions with various examples, we found out that the seemingly orthogonal methods complement each other.

At first, we need to decompose the application using modules, classes, and methods to reflect the hierarchy of domain concepts and provide modularized code which is used in many objects. Then, we add metaprogramming facilities to have modification ability depending on program conditions. And finally, we use holistic manipulation approaches like parse tree modification to change body internals of the core objects.

We then compared two basic implementations. We saw that using interpreter-unaware annotations as comments is an implementation option, but has difficulties in complex feature scenarios. But the DSL approach provides named entities representing features and a minimal syntax to express them at the place they are composed in. Thus, we will further advance the DSL approach.

8 Acknowledgements

Sagar Sunkle is a PhD candidate at the Database group of Otto-von-Guericke University of Magdeburg and receives scholarship from the federal state of Saxony-Anhalt in Germany. Sebastian Günther works with the Very Large Business Applications Lab, School of Computer Science, at the Otto-von-Guericke University of Magdeburg. The Very Large Business Applications Lab is supported by SAP AG.

References

1. S. Apel. *The Role of Features and Aspects in Software Development.* Dissertation, Otto-von-Guericke-Universität Magdeburg, 2007.
2. 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.
3. A. Bryant and R. Feldt. AspectR - Simple Aspect-Oriented Programming in Ruby. Webpage, http://aspectr.sourceforge.net/, last access 10/27/2009, 2002.

4. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, San-Franciso et. Al., 2000.
5. D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O-Reilly Media, Sebastopol, 2008.
6. D. Ghosh. Does Ruby need AOP? Webpage, http://debasishg.blogspot.com/2006/06/does-ruby-need-aop.html, last access 10/27/2009.
7. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing, Crosspoint Boulevard, 2004.
8. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.
9. 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.
10. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, New York, 2008. ACM.
11. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Berlin, Heidelberg, New York, 1997.
12. P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.
13. 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.
14. R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Productline Methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GPCE)*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24, Berlin, Heidelberg, New York, 2001. Springer.
15. R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Techniques. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.
16. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29:127–136, 2004.
17. J. Munnelly and S. Clarke. ALPH: A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL)*, New York, 2007. ACM.
18. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.
19. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11:215–255, 2002.

20. S. Sunkle, S. Günther, and G. Saake. Representing and Composing First-class Features with FeatureJ. Technical report (Internet) FIN-018-2009, Otto-von-Guericke-Universität Magdeburg, 2009.
21. S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. U. Rahman, G. Saake, and S. Apel. Features as First-class Entities - Toward a Better Representation of Features. *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 27–34, 2008.
22. D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragamtic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, 2009.
23. S. Trujillo, M. Azanza, and O. Diaz. Generative Metaprogramming. In *Proceedings of the 6th international conference on Generative programming and component engineering (GPCE)*, pages 105–114, New York. ACM.
24. A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.
25. J. J. VanSlyke. Aspect-Oriented Programming in Ruby with Gazer. Webpage, http://www.elctech.com/projects/aspect-oriented-programming-in-ruby-with-gazer, last access 10/27/2009, 2009.
26. E. Visser. WebDSL: A Case Study in Domain-Specic Language Engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 4143 of *Lecture Notes in Computer Science*. Springer. Tutorial for International Summer School GTTSE 2007, 2008.