Feature-Oriented Programming with Ruby

Sebastian Günther and Sagar Sunkle
Faculty of Computer Science
University of Magdeburg
{sebastian.guenther|sagar.sunkle}@ovgu.de

ABSTRACT

Features identify core characteristics of software in order to produce families of programs. Through configuration, different variants of a program can be composed. Our approach is to design features as first-class entities of a language. With this approach, features can be constructed and returned by methods, stored in variables and used in many expressions of the language. This paper introduces rbFeatures, an implementation of first-class features in the dynamic programming language Ruby. Our goal is to show how such a language extension works with respect to its dynamic host language and the applicability of our results. In particular, we present a step-by-step walkthrough how to use rbFeatures in order to implement known case-studies like the Graph Product Line or the Expression Product Line. Since we created a pure Ruby language extension, rbFeatures can be used with any existing programs and in any virtual machine implementing Ruby.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering - *Design Tools and Techniques*; D.3.3 [**Software**]: Programming Languages - *Language Constructs and Features*

General Terms: Languages

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

1. INTRODUCTION

Software has an inherent complexity. Since the advent of software engineering with the Nato conference in 1968, the question of how to cleanly modularize software into its various concerns is an ongoing question [1]. Today's challenges involve multiple requirements and different domains that software must consider. However, the tyranny of the dominant decomposition forces developers to decompose software along one dimension only [23]. This leads to several software defects, such as tangled and bloated code [13] and structural mismatches of requirements and programs because they can not be mapped one to one. Solution suggested to the code-tangling problems are concepts like Copliens ideas on multi-paradigm design [5], Kiczales et al. about Aspect-Oriented Programming [13] and Prehofer's Feature-Oriented Programming [20].This paper focuses on Feature-Oriented Programming, or short FOP.

Features are characteristics of software that distinguish members of a so called program family [4]. Program families are comparable with Software Product Line (SPL). An SPL is a set or related programs with different characteristics where the features reflect the requirements of stakeholders [11]. The challenge which SPL engineering addresses is to structure valuable production assets in a meaningful way to support productivity and reusability [6]. Withey further defines product lines as a "sharing a common, managed set of features" [25].

Features can be realized with very different approaches: mixin-layers [21], AHEAD-refinements [4], and aspectual feature modules [1] to name a few. Approaches can be differentiated [12] into compositional (e.g., features are added as refinements to a base program [4]) and annotative (e.g., features are implemented as annotations inside the source code [12]).

Because of the several limitations and drawbacks that named FOP approaches have [18, 12, 22], we are actively suggesting new concepts that close the gap between the conceptual and the implementation view of features. One such suggestion was to implement features as first-class entities in a host language. We implemented features as first-class entities with the static programming language Java as the host language called *FeatureJ*[1] [22]. First-class entities are characterized by the following abilities: They can be instantiated at run-time, stored in variables, used as parameters to methods or being returned from them, and also used in various statements and expressions. The gap is closed if we extend a host language with first-class features [22]. In such case features are language-level entities used actively in programming while retaining their status as a high-level modularization concept.

This paper supports our ideas by providing Feature-Oriented Programming (FOP) for the dynamic programming language Ruby. We named our FOP implementation *rbFeatures*[2]. rbFeatures is a language extension for Ruby that provides features as language entities in Ruby. Our motiva-

FOSD'09, October 6, 2009, Denver, Colorado, USA.

[1]http://firstclassfeatures.org/FeatureJ.htm
[2]http://firstclassfeatures.org/rbFeatures.htm

tion is twofold. At first, we want to create another implementation of features as first-class entities to better compare with the Java version and other approaches. We await to gain more insight into language extensions that can be realized with respect to static and dynamic programming languages. The second part is applicability of the result. Since rbFeatures is implemented in pure Ruby, using no external libraries, it is widely usable with both different virtual machines and types of programs. Such discussions however will be deferred to the later part of the paper - the focus is to explain how to use rbFeatures and how features are realized as first-class entities.

The remainder of this paper is structured as follows. The next section will further explain feature-terminology and characteristics. Section 3 introduces the core concepts of the Ruby language. Section 4 explains rbFeatures in detail with a walkthrough by defining an example. We further explain other examples in Section 5. The last two sections discuss our experiences and related work.

2. FEATURE-ORIENTED PROGRAMMING

Modern FOP methods should combine both the conceptual view and the implementation view on features. Considering our earlier proposal to close the gap between the conceptual and implementation views on features [22], we want to elaborate a refined terminology and show properties of first-class features.

2.1 Feature Terminology

We observe that features try to tackle two main problems of software engineering. On the one hand to capture the conceptual higher-level view in order to abstract and configure core functionality, and on the other hand to provide a low-level implementation view for identification and composition of feature-relevant parts of a program.

From a conceptual viewpoint, a feature is an abstract entity which expresses a concern. Concerns can be general requirements of stakeholders [11] or increments in functionality [12]. From the functionality viewpoint, features describe modularized core functionality that can be used to distinguish members of a program family [4]. The functionality viewpoint is taken from here on.

A feature plays an important role in the analysis, design, and implementation phases of software development. In the first part of the development, *conceptual features* describe in natural language, the name, intent, scope, and functions that a feature provides to a program. Conceptual features are thus an additional unit to express the programs decomposition - providing some remedy to the tyranny of the dominant decomposition problem. In the implementation phase, *concrete features* are constructed. Concrete features are the implementation of features inside the program. Often, these concrete features are seen as refinements that add code to a certain base program, e.g. as shown in typical FOP case studies like the graph product line [14] or in the expression product line [15].

This describes our terminology in the remainder of the paper. If not stated otherwise, we use the word feature to denote a concrete feature.

2.2 Feature Properties

Features have properties needed to define their characteristics and role they have in programming. We see properties as defining steps when using FOP. We studied the suggested properties of Lopez-Herrejon [15]. From the viewpoint of dynamic languages, the characteristics *Separate Compilation* and *Static Typing* have no meaning because Ruby is interpreted and typeless. We abstract and generalize the other characteristics as followed (we name the grouping at the end of each property in *italics text*).

- **Naming** Conceptual features are means to abstract the core functionality. This functionality is given a unique name and intent. The concrete feature should also have the same name, while presenting its intent with the following *identification* and *composition* mechanisms. (*Cohesion*)
- **Identification** To form concrete features, one must first identify what parts of the program belong to feature. Those parts can be coarse or fine grained. 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. (*Deltas*)
- **Expressing** A concrete feature needs not only to specify what parts of the program belong to it, but also how these parts are added to the program. Such expressions must work on the same abstraction level as the programs language. (*Deltas*)
- **Composition** Once all feature related code is identified and expressed, the composition is the process to configure and build a variant. It is desirable that the composition-order of features imposes no constraint to compose a working program. (*Flexible Composition*, *Flexible Order*, *Closure under Composition*)

3. RUBY PROGRAMMING LANGUAGE

Ruby is a completely object-oriented dynamic programming language. Its inventor, Yukihiro Matsumoto, took the best concepts of his favorite programming languages and put them inside one language. His foremost motivation was to make programming faster and easier [10].

Ruby has many capabilities for flexible extension and modification of the source code. Naturally, this makes Ruby a good vehicle for FOP. This section introduces Ruby with the most important concepts which play a role in rbFeatures, so that readers yet unfamiliar with Ruby can better follow the ideas proposed later. All material in this section stems from [10] and [24].

3.1 Class Model

Five classes are the foundation of Ruby. At the root lies *BasicObject*. It defines just a handful of methods needed for creating objects from it and is typically used to create objects with minimal behavior. *Object* is the superclass from which all other classes and modules inherit. However, most of its functionality (like to copy, freeze, marshal and print objects) is actually derived from *Kernel*. Another important class is *Module*, which mainly provides reflection mechanisms, like getting methods and variables, and metaprogramming capabilities, e.g. to change module and class definitions. Finally, *Class* defines methods to create instances of classes.

3.2 Core Objects

We discuss the objects of the classes Proc, Method, Class, and Module. The core objects play a fundamental part in rbFeatures. Most of these objects should be familiar to readers experienced with object-oriented programming. However, Ruby's dynamic nature makes following objects more versatile compared to static languages.

- **Proc** A Proc is an anonymous block of code. Like other objects, Procs can either be created explicitly or referenced by a name or implicitly as an argument to a method. Procs allow two kind of usages: On the one hand they can reference variables in their creation scope as a closure[3], and on the other hand they can reference variables which at their creation do not exist. Procs are executed with the **call** method in any place.
- **Method** As a language entity, methods consist of the method's name, a set of (optional) parameters (which can have default values), and a body. Methods are realized inside modules and classes. There are two kinds of Method objects. The normal *Method* is bound to a certain object upon which it is defined. The *UnboundMethod* has no such object, and, for executing it, must be bound to an object of the same class in which the method was originally defined.
- **Class** A class consists of its name and its body. Each class definition creates an object of class Class. 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 modules.
- **Module** Modules have the same structure as classes do. Modules can not have subclassing like inheritance relationships - they can only mix-in other modules.

3.3 Classes instead of Types

Ruby does not have a static language-like type hierarchy for its classes and objects. The now deprecated[4] method *Object.type* would just return the name of the class the object corresponds to. Instead of a type, the classes inside Ruby 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 given 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"[24]. One usage of duck typing is to swap the used objects at runtime, e.g. to provide more performance for data storage.

With this, we finish our introduction of basic concepts. Many more details can be found in [10] and [24]. The next section explains rbFeatures in detail.

4. RBFEATURES

rbFeatures is a language extension to Ruby for realizing Feature-Oriented Programming. Language extension here means two things: At first, new entities and operations are added that represent features at the source-code level, and second that the feature properties defined in section 2.2 to name, identify, express and compose features are supported.

[3]Closures stems from functional programming and capture values and variables in the context they are defined in.

[4]This method was declared deprecated in Ruby 1.8, and has vanished from the current 1.9 release.

To convey our ideas, we choose to present a side-by-side rbFeatures walkthrough. We start with introducing a running example, present the three steps of naming and identifying features, composing a product line and using the product line. Each section motivates an example and explains rbFeatures details. At the end, we summarize the rbFeatures workflow.

For clarity, we introduce two new notations: Slanted text for FEATURES as defined in the product line, and verbatim text for `language entities` which are part of rbFeatures and the developed product line.

4.1 Example: Expression Product Line

The Expression Product Line (EPL) is a common problem used in programming language design as exemplified by Lopez et. al. in [15]. EPL considers the case of different types of numbers, expressions and operations, which are used to compare design and synthesis of variants [15]. The product line is presented as a feature diagram in ►Figure 1. We see that two different NUMBERS, LIT (literal) and NEG (negative literal) exist. The numbers are usable in ADD (addition) and SUB (subtraction) EXPRESSIONS. The OPERATIONS imposed upon an expression are PRINT for showing the string containing the expression and EVAL for evaluating the expression. As can be seen from ►Figure 1, we define the product line with many of its features as being optional, so no strict rules governing the composition need to be considered.

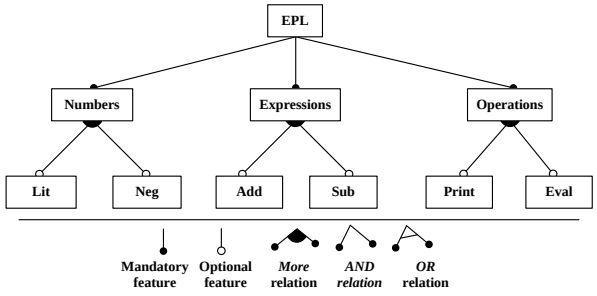

Figure 1: EPL Feature Diagram

4.2 Implementing the Expression Product Line

In three steps we will show how to implement features, define the product line and finally compose and use its variants. Ruby does not impose a fixed structure where expressions must be defined. Programmers can split module and class definitions in separate files, put it all in one file or define it on-the-fly in a shell session using IRB (Interactive Ruby). The original source code for the EPL is contained in one file only.

Step 1: Defining Features

The first step is to define the features. This is done with a very concise notation: objects of class **Class** simply include the `Feature` module. To define the root features, one must write the following (cf. ►Figure 2, left part).

Afterwards, concrete numbers and operations are defined. These features also contain methods themselves. We implement LIT with the following code (cf. ►Figure 2, right part). In line 1 we declare the class `Lit` to be a subclass

```
1  class Numbers
2    is Feature
3  end
4
5  class Expressions
6    is Feature
7  end
8
9  class Operations
10   is Feature
11 end
```

```
1 class Lit < Numbers
2   def initialize(val)
3     @value = val
4   end
5 end
```

Figure 2: Defining basic Features

of `Numbers`. With subclassing, we express the natural tree-structure of the product line. The subclasses here inherit the methods of module `Feature` from their parents. We then define in line 3 the `initialize` method which receives the initial value for `Lit`. We implement `Neg` likewise.

This is followed by defining `Add` with the following code (cf. ►Figure 3). We use again subclassing to define the relationship to `Operations}`, and define the `initialize` method to get two values (which can be `Add`, `Sub`, `Neg` and `Lit` values). Again, `Sub` is implemented likewise.

```
1 class Add < Operations
2   def initialize(left, right)
3     @left = left
4     @right = right
5   end
6 end
```

Figure 3: Defining the Add Operation

Having defined all `Numbers` and `Expressions`, we now define the `Print` feature (cf. ►Figure 4). We then add `print` methods to `Lit` and `Add`. Line 5 shows a *feature containment*.

```
1  class Print < Operations
2  end
3
4  class Lit
5    Print.code do
6      def print
7        Kernel.print @value
8      end
9    end
10 end
```

Figure 4: Defining Print Feature and Operations

The first part, just `Print` in this case, is the *containment condition*. A containment condition is an arbitrary expression combining features with operations. The second part is the *containment body*. The body is surrounded by a **do...end** block, and contains in this case a method definition for `print`. The containment body can be any Ruby statement. Feature containments can be used to encapsulate coarse-grained modules, classes, and methods, as well as fine grained individual lines or even parts of a source code line. In total, the containment from line 5 to 9 expresses that only if `Print` is activated, then the `print` method is defined properly. Likewise, implementing the EVAL feature is expressed as a feature containment with `Eval` as the containment condition, and a declaration of the `eval` method (which returns the value of `Lit`) as the containment body.

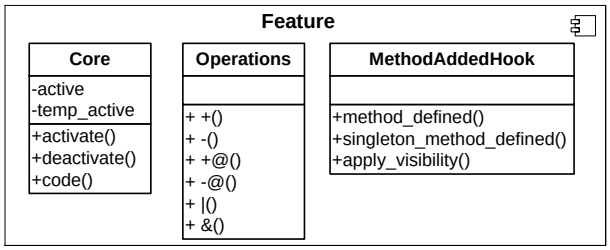

Figure 5: The Feature Module

The internals of features are as follows. rbFeatures defines the `Feature` module as a composition of three different modules, as seen in ►Figure 5. Each module encapsulates a set of operations needed for `Feature` to function properly:

- **Core** Provides the basic method to use a `Feature` as a configuration unit. We spoke of activating a feature - meaning that the internal class-variable `active` is set to true. The methods `activate` and `deactivate` change the activation status. The other operation is `code`, which was used in the example for defining the `print` method. The `code` receives a `Proc` object that is the containments body. Internally, `code` checks if the feature is activated - if yes, the body gets executed.
- **Operations** This module defines operations[5] which can be used inside the containment condition:
  - Plus (+) => Activation (single feature)
  - Minus (-) => Deactivation (single feature)
  - And (&) => Activation (all features)
  - Or (|) => Activation (at least one feature)

  Operations and features can be chained to arbitrary expressions. Conditions like "If feature A, B and C, but not D are activated", translate to a natural syntax `(A + B + C - D)`. Or "Feature A or B and C, but not D" translates to `((A | (B & C)) - D)`.
- **MethodAddedHook** This module defines a special hook. When methods are defines in an object, the private method *method_added* is called. We use this as a hook in the case of a feature containment defining a method. If the containments condition is not true, then the methods body will be replaced with a error message. This error message details which features activation status prohibits the method's definition. For example, if the `print` method is called on a `Lit`, but feature `Print` is not activated, the error message will be "*FeatureNotActivatedError: Feature Print is not activated*". This error message is computed dynamically and names the right-most feature inside a containment condition that has the wrong activation status. The `apply_visibility` method is needed to retain the visibility of the defined method in the case its body is overwritten.

[5]Concerning the method-declaration as shown in ►Figure 5: The methods containing an "@" symbol are unary operations defined on the object itself.

Step 2: Implementing the Product Line

Assume that all other features and the concrete semantics of each entity have been implemented. We now need to implement the Product Line. Currently, we use no special object, but a native `Proc` object. This allows two composition approaches: To manually surround all code in a `lambda do; ...; end` block (cf. ►Figure 6, line 1 to 5) or to join the content of all source files and define a lambda from them. To stay at a higher abstraction level, we usually name this proc as the product line it represents.

```
1 EPL = lambda do
2 class Lit < Numbers
3   def initialize(value)
4 ...
5 end
6
7 FeatureResolver.initialize EPL
```

Figure 6: Defining and Initializing the EPL

Having defined the product line, the user then directly interacts with a module called `FeatureResolver`. This module handles initialization and resetting of the product line. As shown in ►Figure 7, four methods are available. We explain in opposite direction. With `reset!`, all classes which include the `Feature` module are deleted so that their initial definition can be restored. The `update` method is called whenever a feature changes its activation status. Finally, `init` and `register` are used during initialization.

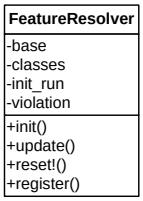

Figure 7: The Feature Resolver Module

The initialization composes the product line by evaluating the proc. Evaluating means that all code is executed once, and this leads to classes and modules defining the program. Classes including the `Feature` module register themselves with the `FeatureResolver`. A challenge is to regard classes or methods defined in feature containments. Most feature containments will not be executed because one feature may violate the containment condition. As mentioned before, rbFeatures informs the user if a method can not be called because of a certain feature violating the containment condition. Methods may be defined in containments which are violating a condition. How to define those methods properly? In the initialization phase, all containments are executed once - even if a violation occurred. This can lead to a method declaration, which is caught by the `MethodAddedHook`. If a violation occurred and the initialization phase is happening, then the original method body will be replaced with the error message. In the case of features defining functions themselves, their methods get deactivated with the same mechanism. Once the product line is composed, we can start using it actively.

Step 3: Composing and Using Variants

After initialization, the EPL is ready to use. The dynamic nature of Ruby allows two kinds of usages: static configuration of a variant before executing the program and dynamic configuration at runtime.

We start with static configuration. By defining the activation or deactivation of certain features, we define a variant. For example, we want to define a EPL with features Lit, Add, Sub and Print. This configuration can be represented as a class containing activation statements (cf. ►Figure 8).

```
1 class Variant1
2   Lit.activate
3   Add.activate
4   Sub.activate
5   Print.activate
6 end
```

Figure 8: Static Configuration of EPL

```
1  >> FeatureResolver.init ExpressionProductLine
2  => true
3  >> Add.activate
4  => :activated
5  >> Lit.activate
6  => :activated
7  >> Lit 1
8  => #<Lit:0xb7b35968 @value=1>
9  >> a = Add Lit(11), Lit(7)
10 => #<Add:0xb7c57544 @right=#<Lit:0xb7c57558
       @value=7>, @left#<Lit:0xb7c5756c @value=11>
11 >> a.print
12 FeatureNotActivatedError: Feature Print is not
       activated
13
14 >> Print.activate
15 => :activated
16 >> a.print
17 11+7=> nil
18 >> a.eval
19 FeatureNotActivatedError: Feature Eval is not
       activated
20 >> Eval.activate
21 => :activated
22 >> a.eval
23 => 18
24 >> Print.deactivate
25 => :deactivated
26 >> a.print
27 FeatureNotActivatedError: Feature Print is not
       activated
```

Figure 9: Session with EPL

The second usage kind is dynamic configuration. At runtime, we can activate and deactivate arbitrary features. Consider the following session with the interactive Ruby shell (►Figure 9). In the session, lines starting with "`>>`" denote input, and lines with "`=>`" denote output. In line 2 we load the `ExpressionProductLine` into the `FeatureResolver` module. Following lines 4 - 7 activate the features Add and Lit. We then create a single Lit object (line 8), and a compound Add object (line 10). Line 11 shows the return value of object creation - its in-memory representation. We then want to call `print` on the add statement (line 12), but get a `FeatureNotActivatedError` in return. After activation in line 15, we can print and see `11+7` in line 18. The same is shown for `eval` in lines 19-25. After that, we decide to de-

activate `Print` again by calling the same-named method in line 26. Successively, calls to `print` fail again.

4.3 Summary Workflow

Summarizing, using rbFeatures consists of the following steps. ►Figure 10 shows these steps graphically.

- Name and identification of the features
- Defining features via including the feature module and building subclass relationships
- Put all source code inside a Proc
- Initialize the proc to compose a variant
- Static or dynamic configuration of the variant

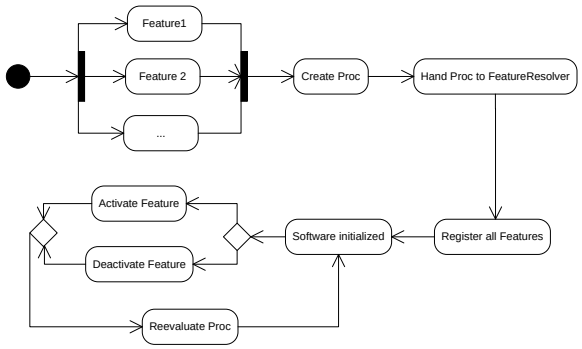

Figure 10: rbFeatures' basic Workflow

5. OTHER EXAMPLES

The EPL illustrates rbFeatures with a simple scenario. We also developed two other programs which are presented shortly. Our major motivation is to show that rbFeatures was successfully applied to two other product lines, with the second example having a graphical representation as well.

5.1 Graph Product Line

The Graph Product Line (GPL) describes a family of related program in the domain of graphs [14]. We see a graphical representation of the features and their relationships in ►Figure 11. We assume the reader understands this figure with respect to the legend in ►Figure 1.

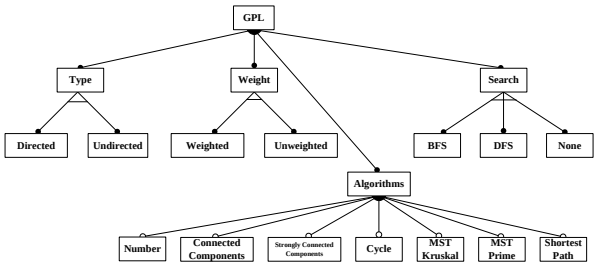

Figure 11: The Graph Product Line

Although heavyweight graph computations demand a solution using matrices, we implemented all entities of the domain, like nodes and edges, as objects. This was also reported in [14] as a better way to compose the program. Our approach was to first develop the whole program without thinking about features. Once all algorithms were implemented, we feature refactored the program. Refactoring means to define concrete features with the same name as shown on the feature diagrams, and then to identify and express those parts of the program which belong to a certain feature.

An example of the refactoring is shown in ►Figure 12. The feature Weight modifies the `Edge` class by forming containments around the `attr_accessor` (line 3) and by adding another line which deletes any given weights to edges so that the body is not changed (line 5).

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

Figure 12: Feature-Refactoring Changes in GPL

Feature containments for Type only wrapped existing lines. Finally, all Algorithms and Search features are put in containments. No further changes are required - even if features depend on other features. Consider the case of using Strongly Connected Components. It requires to use DFS. If a variant is created without activating DFS, then calling the method `strongly_connected_components` simply returns a "*FeatureNotActivatedError: Feature DFS is not activated*". Activating DFS remedies this situation.

5.2 Calculator Product Line

In the third example, we targeted an open source implementation of a simple graphical calculator. Feature refactoring changes to the source code were of medium nature, because the original program was very concise and used batch-like operation for method declaration. We also added a graphical configurator. At runtime, multiple instances of the Calculator, configured in different variants, can be created (►Figure 13).

6. COMPARISON AND DISCUSSION

In order to integrate rbFeatures into overall FOP research, we want to discuss a number of points. In our view, the most distinguishing point are to realize FOP as a pure and lightweight language extension, to use a dynamic programming language and finally the possibility to include modeling capabilities for further closing the gap between abstract and concrete features.

Language Extension

Many techniques have been introduced that implement FOP [2, 3, 17, 19]. Features are represented in terms of refinements, feature structure trees, more flexible containers than classes and interfaces, and hyperspaces. These approaches either impose modifications to the existing tools used in writing and generating the programs, like compiler extensions,

Figure 13: Configuring and Executing Variants of the Calculator Product Line

or use an external structure which maps features to an existing source code. rbFeatures has no such restrictions.

First, it is based on pure Ruby - no change to the interpreter and no additional library is needed. Programmers just need a virtual machine implementing Ruby 1.8.6 respective 1.9, such as MRI[6] written in C or JRuby[7] written in Java. The VM's capability may further broaden what programs can be written using rbFeatures. JRuby, which uses Java as its interpreter language, allows many cross-language developments, like calling Java code from inside Ruby or vice versa [9]. As applications written in JRuby can access native Java libraries, those libraries become feasible for FOP using rbFeatures.

Second, rbFeatures operates on the same abstraction level as the program - the language level. Developers directly include feature containments at the place in the program where code belonging to a feature is expressed. They do not need to switch to another representation, possible using another language or syntax. In terms of the feature properties in section 2.2, the tasks of feature identification and expression become the same in rbFeatures. Further more, the amount of changes to a program without features is very minimal. Typical changes just introduce containments with a condition around an existing block of code. At least with the presented case studies, we seldom had a case where we need to rewrite the code. ►Table 1 shows what changes occurred in the show examples when rbFeatures was used. Column 2 and 3 lists the loc for the normal and feature-based version of the program. Column 4 shows the total LOC added, followed by percental increase. We see that smaller programs tend to have a comparatively big amount of changes when using rbFeatures. For programs with a large LOC, the lines added from using rbFeatures are relatively small. Also, the number of methods is an indicator for changes. For example, the changes in EPL were mostly the containments `do...end` block around the methods.

Dynamic Language vs. Static Language

Techniques implemented in statically typed object-oriented programming languages have dominated research in FOP. Most of the aforementioned techniques are implemented in more conventional programming languages like Java and C++, which are usually static. On the contrary, Ruby's dynamic typing and interpreted execution enables metaprogramming mechanisms which strengthens implementing language extensions. Proc objects defined at a certain place in the program can either be called at this place or called in another context. And since the variables contained inside a proc can either reference existing ones or yet to be defined ones allows very flexible composition. But this flexibility comes at a price. First, static languages are more efficient then interpreted languages in terms of runtime and performance. Second, typeless languages can introduce bugs to programs that are only countered with testing. We used a fully test-driven approach to rbFeatures and tested all mentioned properties in different contexts.

[6]http://www.ruby-lang.org/en/
[7]http://jruby.codehaus.org/

PL	Normal	Feature	#LOC	%LOC	Methods
EPL	74	97	23	31%	10
CPL	70	89	19	27%	4
GPL	291	323	32	11%	9

Table 1: Comparing LOC for rbFeatures Programs

Including Feature Modeling Capabilities

rbFeatures can be seen as a so-called Domain Specific Language (DSL). It uses suitable notation and abstraction to represent domain-knowledge and concepts in a precise form [8]. For rbFeatures, this domain is feature-oriented programming. But, we can extend rbFeatures with concepts from textual feature modeling languages to allow expressing both conceptual features and concrete features.

The array of feature modeling languages is vast - we want to give two examples here. Deursen and Klint propose a feature description language in [7] in which they consider automated manipulation of feature descriptions. Feature composition is achieved by translating the textual feature descriptions to unified modeling language models. Similarly, Loughran et al. [16] present the variability modeling language (VML). VML supports first-class representation of architectural variabilities. VML consists of explicit references to variation points and composition of both fine and coarse-grained variabilities. With this capability, VML resembles more an architectural description language.

Expressing conceptual features in rbFeatures requires a language extension which uses similar metaprogramming concepts as shown before. We would need a first-class representation of a product line, product variant, and constraints regarding legal combinations of features. This is future work.

7. SUMMARY

rbFeatures is a pure language extension to Ruby in order to enable feature-oriented programming. We introduced the Ruby language, and presented a step-by-step walkthrough how to use rbFeatures and how the extension works internally. By discussing the main cases studies of FOP, namely the Graph Product Line and the Expression Product Line, we enable the direct comparison of rbFeatures with other approaches.

Defining features as first-class entities leads to full usability of features in all parts of a program. Features can be stored in variables, returned from methods and extended like any other object. Only minimal changes are required to make a program feature-oriented. Because of the lightweight implementation using only core Ruby mechanisms, rbFea-

tures is useable with all other virtual machines.

We want to extend rbFeatures in several ways. Additional to known FOP case studies, we want to use rbFeatures also in the context of web applications written with the Rails framework. Furthermore, we want to combine rbFeatures with a already implemented software product line configuration language for abstract modeling and concrete implementation at the same abstraction level.

Acknowledgements

We thank Christian Kästner for his comments on an earlier draft of this paper.

8. REFERENCES

[1] S. Apel. The role of features and aspects in software development. PhD Thesis, Otto-von-Guericke-Universität Magdeburg, 2007.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, Springer, Heidelberg, 3676:125–140, 2005.

[3] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, pages 702–703, 2004.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.

[5] J. O. Coplien. Multi-paradigm design. PhD Thesis, Vrije Universiteit Brussels, 2000.

[6] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, Boston, San Francisco et. al., 2000.

[7] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(6):1–17, 2002.

[8] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[9] J. Edelson and H. Liu. *JRuby Cookbook*. O'Reilly Media Inc., Sebastopol, 2008.

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

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. *Technical report CMU/SEI-90-TR-021*, Carnegie Mellon University, 1990.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, ACM, New York, pages 311–320, 2008.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg et. al, 1241:220–242, 1997.

[14] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2186:10–24, 2001

[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)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 3586:1–37, 2005.

[16] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Proceedings of the 7th International Symposium on Software Composition (SC)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg 4954:36–51, 2008.

[17] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. *ACM Press*, pages 211–222, 2001.

[18] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[19] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, IEEE Computer Society, pages 729–730, 2001.

[20] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1241:419–443, 1997.

[21] 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 (TOSEM)*, 11(2):215–255, 2002.

[22] S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. Ur Rahman, G. Saake, and S. Apel. Features as first-class entities - toward a better representation of features. In *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 27–34, 2008.

[23] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, ACM, pages 107–119, 1999.

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

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