Science of Computer Programming 77 (2012) 152–173

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

rbFeatures: Feature-oriented programming with Ruby

Sebastian Günther *, Sagar Sunkle

Faculty of Computer Science, University of Magdeburg, Germany

ARTICLE INFO

Article history:
Available online 1 February 2011

Keywords:
Feature-oriented programming
Domain-specific languages
Dynamic programming languages

ABSTRACT

Features are pieces of core functionality of a program that is relevant to particular stakeholders. Features pose dependencies and constraints among each other. These dependencies and constraints describe the possible number of variants of the program: A valid feature configuration generates a specific variant with unique behavior. Feature-Oriented Programming is used to implement features as program units. This paper introduces rbFeatures, a feature-oriented programming language implemented on top of the dynamic programming language Ruby. With rbFeatures, programmers use software product lines, variants, and features as first-class entities. This allows several runtime reflection and modification capabilities, including the extension of the product line with new features and the provision of multiple variants. The paper gives a broad overview to the implementation and application of rbFeatures. We explain how features as first-class entities are designed and implemented, and discuss how the semantics of features are carefully added to Ruby programs. We show two case studies: The expression product line, a common example in feature-oriented programming, and a web application.

1. Introduction

Software is inherently complex. 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 still an ongoing question [1]. Today's challenges include multiple requirements and different domains that software must consider. However, the tyranny of the dominant decomposition forces developers to implement software along one decomposition dimension only [54]. This leads to several software code smells, such as tangled and bloated code [41]. Solutions suggested for these problems are for example multi-paradigm design [20], aspect-oriented programming [41], and feature-oriented programming [46]. This paper focuses on feature-oriented programming, or short FOP.

In software engineering, features provide an additional dimension of modularization to applications. Features can be seen from a conceptual viewpoint and from an implementation viewpoint. *Conceptual features* represent the requirements of stakeholders in the context of a software product line [34]. A particular feature configuration is used to distinguish individual members or variants [11]. Product lines share a "common, managed set of features" [59] and other valuable production assets, such as documentation, that are structured in a meaningful way to support productivity and reusability [22]. Software is realized as source code, and so the conceptual features must be expressed as or inside code. *Implementation features* can be realized with very different FOP approaches such as mix-in layers [52], AHEAD refinements [11], and aspectual feature modules [1]. In general, FOP approaches can be differentiated into compositional (features are added as refinements to a base program [11]) and annotative (features are implemented as annotations inside the source code [37]).

The current approaches face several challenges. They suffer from the dependence on the feature composition order, the inability to address code fragments of varying granularity, the lack of integrated type checking, and the representation mismatch between feature modeling and implementation approaches [45,37,53].

* Corresponding author.
E-mail addresses: sebastian.guenther@ovgu.de (S. Günther), sagar.sunkle@ovgu.de (S. Sunkle).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.12.007

We think that these difficulties are coming from the gap between the conceptual features and their corresponding implementation in a program. Our approach is to implement the entities used in feature-oriented programming (features, product lines, and variants) as first-class entities of a given host language. This approach provides several advantages that reflect the nature of first-class entities. First, these entities are abstractions that extend a host language's capabilities. Second, these entities are mutable objects with data and behavior that can be further modified at runtime. Third, first-class features allow one to reason about a program's structure that exceeds common modularization mechanisms such as methods and classes. Fourth, the entities are part of the language and can be used in conditionals and functions that process (or modify) the entities' data.

Our approach allows to obtain a product line model and a feature-refactored representation of the product line code as abstractions of a program. These abstractions are used to validate feature configurations as specified by the product line model and to generate variants with specific configuration. At runtime, variants and the product line model can be changed too. Therefore, the gap between conceptual features and their implementation is effectively closed. In an earlier publication, this approach was shown for the Java programming language [53]. In this paper, we are presenting rbFeatures, a pure Ruby language extension.

rbFeatures uses metaprogramming capabilities and support for functional programming to implement features and product lines as first-class entities. The essence of rbFeatures is to use *feature containments* to semantically annotate those parts of a program that belong to a particular feature. Containments encapsulate Ruby code of coarse granularity, such as modules and classes, as well as fine granularity, such as lines in method bodies or even single characters in a source code line. Whether a feature containment is executed is determined by the containment condition, a logical formula that expresses which single or compound features needs to be active.

The initial implementation of rbFeatures, documented in [28], focused on implementation features. Since then, we succinctly refined rbFeatures to provide a first-class representation of complete software product lines and their variants, enabling powerful and safe[1] run-time adaptation and composition of product lines and their individual variants. rbFeatures' code manipulation mechanisms are carefully designed not to interfere with the base program logic. This enables rbFeatures to be used in combination with all other Ruby programs, frameworks, or domain-specific languages, as discussed in [27].

The remainder of this paper is structured as follows. Section 2 provides required background information on FOP, software product lines, and the relevant objects and mechanisms of the Ruby programming language. Section 3 introduces rbFeatures first-class entities, the used metaprogramming concepts, and the three ways how rbFeatures can be used. The following two sections explain case studies: Section 4 the expressions product line, and Section 5 a web application. Section 6 contrasts our approach to related work, and Section 7 summarizes the paper. We use the following fonts: *keywords*, `source code and language entities`, and FEATURES.

2. Background

2.1. Feature-oriented programming

Feature-Oriented Programming aims at implementing an application in terms of a set of separate features that can be activated and deactivated as needed. Each set of activated features yields a special application variant with unique behavior. We observe that features try to tackle two main problems of software engineering: On one hand to provide the conceptual higher-level view for abstracting and configuring 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 [34] or increments in functionality [37]. From the functionality viewpoint, features describe modularized core functionality that can be used to distinguish members of a program family [11]. Both viewpoints lead to (presumably) separate concepts of what a feature is when regarded in the analysis, design, and implementation phases of software development. In analysis and design, conceptual features describe in natural language the name, intent, scope, and functions that a feature provides to a stakeholder. Conceptual features are thus an additional unit to express the programs decomposition — limiting the effect of 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.

This paper proposes a unified view of conceptual and concrete features. Summarizing the article [43] and adding our own experience, we see that using such unified features in software development requires the following steps:

- *Naming* — Features are entities to abstract core functionality, and their name expresses this functionality.
- *Identification* — Concrete features encapsulate coarse or fine-grained parts of an application [37]. 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 modules. Fine-grained features (also called cross-cutting features) impact various parts of the source

[1] Ruby has no type system, but by providing a product line model with constraint rules that express valid feature configurations only correct configurations are eligible. Additionally, Ruby programs rely on extensive test-suites to ensure correct behavior, which is recommended to be provided for feature-refactored programs too.

code: Extending code lines around existing blocks of code, changing parameters of methods, or adding single variables at arbitrary places. The identification step analyzes the application and marks code relevant for a particular feature or a combination of features.

- *Representation* — Identified parts of the application need to be represented by either (a) using explicit or implicit annotations of the source code, (b) using modularization concepts of the host language, or (c) by specifying feature-related code in an external format that is composed with the base application.
- *Composition* — Composition is the process to configure and build a variant of the product line by combining the parts belonging to features with the basic program.

These steps are applied independently of whether the program is developed with features in mind or whether an existing program is feature-refactored. Developers require a structured approach to support these steps. We will see that rbFeatures is such an approach.

2.2. Software product lines

Software development can be broadly distinguished into the design of one-off systems, which are developed for one specific purpose only, and program families or product lines, which share a common code base and other assets. Software product lines show a conceptual analogy to mechanical assembly lines, in which similar parts are produced but assembled to different products [22].

Today's needs to individualize and customize software has a strong economic background because "managers must invest strategically in software assets to gain competitive advantage in the battlefield or the marketplace" [59]. Software product lines techniques help to identify, structure, and provide production assets such as program documentation, configuration, source code, libraries and more of the product line [22]. Assets can be common to all products or variant-specific. The common assets provide the base for all variants, therefore evolving the common parts of the software also evolves all variants. Feature configuration constraints are usually defined in a feature diagram such as the notation shown in [22]. Different features of the program can have mandatory, optional, or strict constraints imposed on other features, thus limiting the amount of possible combinations. For structuring the code base to represent features, FOP can be used.

2.3. Ruby programming language

Ruby is a completely object-oriented and interpreted dynamic programming language. A Ruby program can be composed by reading source code files or by evaluating strings with Ruby code that can come from multiple sources. Several interpreters for Ruby exist. The two most mature ones are the original MRI[2] written in C and JRuby[3] written in Java. A complete language specification is available online.[4]

Ruby has many capabilities for flexible extension and modification of programs even at runtime. 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. The information for this section stems from [25,55].

2.3.1. Class model

Five classes form the root hierarchy of Ruby.[5] At the root lies `BasicObject`. It defines just a handful of methods 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 the `Kernel` module. 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` is a metaobject with which all other classes are created. Fig. 1 summarizes the relationships.

Ruby's classes and modules are defining the namespaces of the program. Ruby also separates four different scopes in which code can be executed: Top-level, in modules or classes, in method declarations, and finally inside `Proc` objects (see below). Therefore, the namespaces introduced by module and class declarations also form a scope in which code can be executed, for example to add additional modules or classes.

2.3.2. Core objects

We discuss the classes `Module`, `Class` and `Proc` that have a major role in rbFeatures.

- *Module* — Module declarations consist of a name and a body. They are used to group a set of related methods which are mixed-into other modules, classes, or class instances.

---

2 http://www.ruby-lang.org/en/.

3 http://jruby.codehaus.org/.

4 Currently as a specification draft from http://ruby-std.netlab.jp/draft_spec/draft_ruby_spec-20091201.pdf. Additionally, the Ruby community provided a complete set of executable tests to help in implementing Ruby interpreters at http://rubyspec.org/wiki/rubyspec.

5 `BasicObject` was added to Ruby 1.9. In prior versions, `Object` is the root entity.

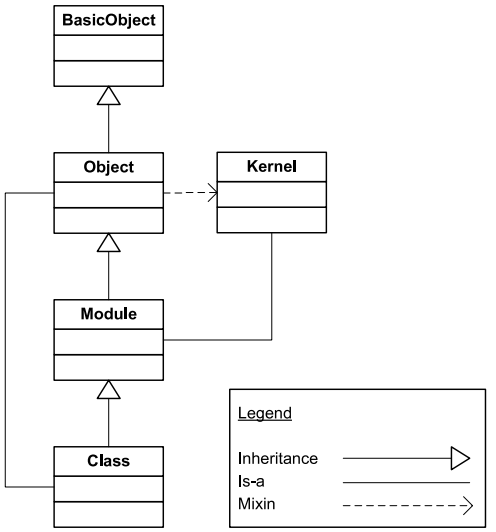

Fig. 1. Ruby's class model.

- *Class* — Class declarations also consist of a name and a body. In contrast to modules, classes can form a hierarchy of related classes via single inheritance, gaining all methods of their parents. Methods and variables can be defined for the classes themselves or for the instances created from the classes with calling `new`.
- *Proc* — A proc is an anonymous block of code. Like other objects, Procs can either be created explicitly (with expressions `Proc.new` or `lambda`) or implicitly (argument to a method using a special syntax). Procs can be used in two ways. On one hand they can reference variables in their creation scope as a closure.[6] And on the other hand, they can reference variables which are provided in the scope where the Proc is executed, thus deferring the concrete Proc's semantics to the execution moment. Procs can be executed with the `call` method anywhere in the program.

Ruby has two additional properties that are important for rbFeatures. First, Ruby's type system is described as "duck-typing": objects just need to provide the methods called upon them to be used in different places of the program. Second, Ruby provides powerful reflection and metaprogramming facilities[7] to obtain the runtime structure of programs and to add and delete the behavior of all objects. Even Ruby's core classes, like `String` or `Array`, can be modified at runtime.

3. rbFeatures

rbFeatures is a domain-specific language (DSL) for FOP. DSLs are defined as providing the suitable abstractions and notations for the domain [57,30] as well as being specifically tailored for a domain [33]. rbFeatures provides these characteristics by implementing first-class entities for conceptual and implementation features, software product lines, and variants, as well as the required semantics to change and extend product lines. This section details the corresponding first-class entities, the background entities governing the program initialization and modification, the used metaprogramming facilities, and the usage kinds of rbFeatures.

3.1. First-class entities

There are four first-class entities in rbFeatures: `Feature`, `FeatureModel`, `ProductLine`, and `ProductVariant`. They are called first-class because they are high-level abstractions that represent complete product lines and variants that can be used throughout the program and can be integrated with the program's semantics. The entities' implementation is depicted in Fig. 2 in the form of a class diagram, and their relationships are shown in Fig. 3. The following paragraphs explain these entities conceptually, concrete examples are presented in the case studies in Sections 4 and 5.

3.1.1. Feature

Implementation features are realized by classes that mix-in the `Feature` module. By doing so, the class is extended with several methods that implement the semantics needed to express which code of the program belongs to the feature. Such representations are called *feature containments*, they consist of a *containment condition* and a *containment body*. The body, which is expressed as a `Proc` object, is passed to the feature's `code` method. The body can contain coarse-grained entities,

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

[7] See our earlier paper [28] for an overview about Ruby's metaprogramming capabilities that enable FOP.

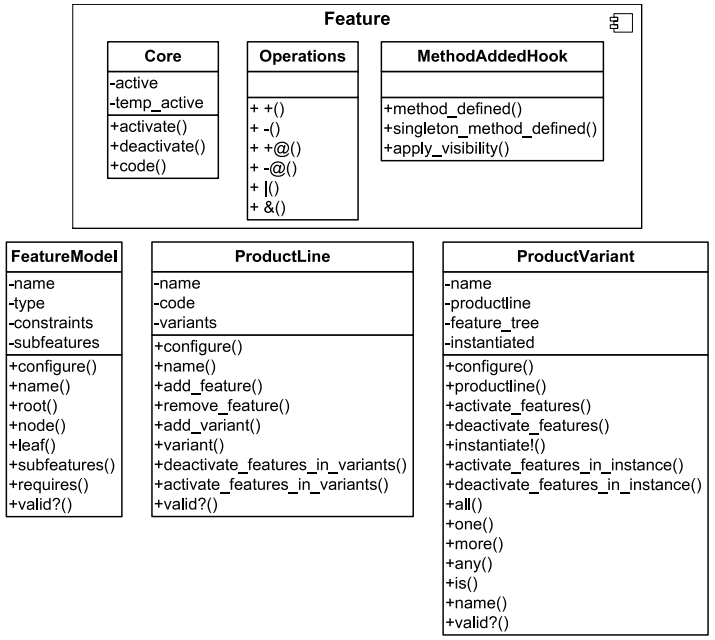

Fig. 2. rbFeatures: overview of the first-class entities' implementation.

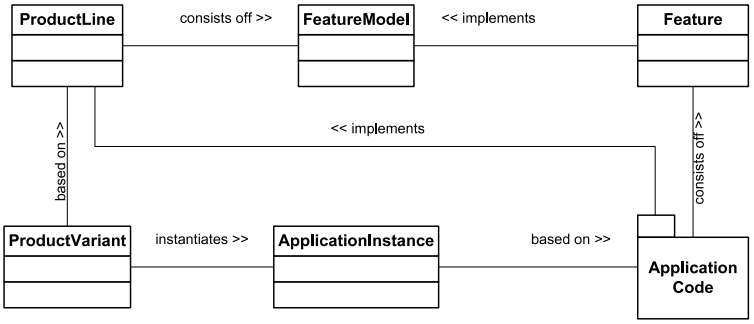

Fig. 3. rbFeatures: entity structure.

such as modules and classes, as well as fine-grained fragments, such as individual lines or even single characters inside code lines. The condition determines which feature configuration leads to the execution of the body. The condition can be a single feature or a complex logical condition between several features. For example the expression "Feature A and B but not C, or D, must be activated" translates to the condition `(A + B ! C) | D`.

The `Feature` entity internally consists of three modules to factor out different behavior as separate yet composable entities. Each module contributes with unique functionality:

- *Core* — Provides the basic methods to use a `Feature` as a configuration unit. The methods `activate` and `deactivate` change the feature's activation status by modifying the `@active` instance variable. The other operation is `code` which receives a `Proc` object that is the containment's body.
- *Operations* — This module defines operations[8] which can be used inside the containment condition. The containment condition is satisfied if all its operands and operations satisfy the following conditions
  - Plus (+) => The second operand must be activated (can be used in postfix notation too).
  - Minus (−) => The second operand must be deactivated (can be used in postfix notation too).
  - And (&) => All operands must be activated.
  - Or (|) => At least one operand must be activated.
- *MethodAddedHook* — This module defines a special me-taprogramming hook that is used in the initialization of the program. This is explained in detail in Section 3.3.

8 Concerning the method declaration as shown in Fig. 3: The methods containing an "@" symbol are unary operations defined on the object itself.

3.1.2. FeatureModel

Conceptual features are realized with the help of the `FeatureModel` class. The class is not instantiated directly, but its `configure` method is used to declaratively express[9] the following properties of a conceptual feature:

- The position in the feature tree with the methods `root`, `node`, and `leaf`.
- The name of the feature with the `name` method.
- The constraints of the feature are expressed with the `require` method. Keywords like `one` or `more` express that a feature requires another feature mandatorily or optionally.

3.1.3. ProductLine

The product line is represented as a class that collects a number of conceptual features. It is created with the `configure` methods, receives a `name`, and a set of string representations for the `code` of this product line. Already created `FeatureModel` entities are added via the `add_feature` methods. The product line also stores a list of the available `variant` objects, and it provides methods to alter the feature configuration in all of its associated product variants like, for example `activate_features_in_variants`.

3.1.4. ProductVariant

Variant objects include a reference to the originating `productline` and have a unique `name`. Once created, they allow to set a specific feature configuration with `activate_features` and `deactivate_features`. The results of these operations are stored in the local `@feature_tree` variable. Up to here, the variant object just represents the feature configuration. Calling `instantiate!` will trigger the following steps and eventually result in an instantiated product:

1. Validate the feature configuration (see below).
2. Create a module with the name of the variant.
3. Evaluate the product line code in the scope of the module, thereby declaring the programs modules, classes, and variables in the scope of this module.
4. Activate all features in the instantiated variant according to the feature configuration.

Once instantiated, the variant also provides methods to update its instance with a new configuration, but each configuration is also validated before modifying the instance.

3.1.5. Validations

Along the chain of creating the first-class entities `ProductVariant` – `ProductLine` – `FeatureModel`, following validation steps happen:

- `ProductVariant` — A variant is valid when the particular feature configuration does not conflict with the feature constraints of its product line. Variants can be built only when the product line itself is valid.
- `ProductLine` — A product line is valid (1) when the feature tree only has one root feature, (2) when the features have a correct position among its subfeatures (a leaf cannot have children), (3) when all features mentioned in the subfeatures and all features mentioned in the constraints are included in the `ProductLine`, (4) and when there are no unconnected nodes in the tree. The `ProductLine` refuses to add invalid `FeatureModel` to its internal representation.
- `FeatureModel` — Is valid if it has a `name` and a `position` property.

3.2. Helper entity

The `FeatureResolver` module, shown in Fig. 4, is the helper entity that handles updates of the product line code due to changes of the features' activation status. In the initialization of the program, all source code parts (stemming from one or several files) are added to the `FeatureResolver` and stored internally.[10] Once all parts are added, the program is instantiated with the `init` method that leads to the declaration of the program's modules, classes, and methods.

After initialization, features can be activated and deactivated to change the program. All changes trigger the execution of the `update` method. This method determines the product line of the feature which was updated, and then re-executes the code that is stored for this product line. Eventually new feature containments have become active because their condition is satisfied. Executing the containments can add new behavior. Finally to set the program to its initial state with all features deactivated, the `reset!` method can be used.

3.3. Metaprogramming facilities

Having explained all entities, this section details which kind of Ruby's metaprogramming facilities are used in rbFeatures. We explain the facilities as they are used in the initialization and usage of a rbFeatures program.

---

[9] This notation stems from a DSL we developed earlier [26].

[10] In the case of using a `ProductLine` (see usage kinds explained in Section 3.4), the source code is stored as the `code` object in the `ProductLine`.

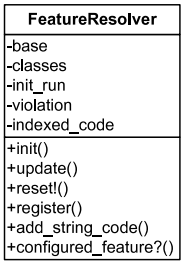

Fig. 4. rbFeatures: the `FeatureResolver` entity.

3.3.1. Variant instantiation

Variants are representations of a particular feature configuration. Calling `Variant.instantiate!` will actually instantiate the variant. The instantiation uses *runtime code evaluation* for this.

> **Runtime code evaluation** — At runtime, a Ruby program can be modified by evaluating code in two forms: Stored as internal `Proc` object, which is immutable to changes,[11] or in the form of a string, using built-in string manipulation operations for arbitrary changes. Using the `eval` method, such code representation can be executed inside an arbitrary namespace or inside the four mentioned scopes (see Section 2.3.1). There is no limit to the content of such code: Class or module declarations, overriding methods, or even accessing data from within an object. Practically, string objects are more open to customization especially because the scope they are executed in can be composed dynamically (such as to use reflection to read a hierarchy of modules and classes to execute code at a very specific point). `Proc` objects instead act as closures that catch the binding of variables in the surrounding scope.

When the variant is instantiated, a custom `String` object is created that consists of a module declaration with the name of the variant and the product line code as the module's body. Evaluating this string creates the module and in its scope the application's modules, classes, methods, and variables.

3.3.2. Initial product line program execution

Feature containments can encapsulate whole classes and modules. Some of them may be required for the correct execution of the program. To ensure the correct program execution, all feature containments must be executed once to define all program entities. However, if containments with an unsatisfied condition include method declarations, the methods are modified with the *method added hook*.

> **Method added hook** — When methods are defined in an object, the private method `method_added` is called. By overriding this method, custom behavior can be implemented.

The method added hook modifies the method's behavior to return an error message. This error message details which features' activation status prohibits the execution of the declared method. For example, if the `print` method is called on the `Lit` class, 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 containment's conditions failing feature.

3.3.3. Feature activation and deactivation

After the initial startup of the application, the feature configuration can be changed at runtime. To support the modification of the currently running program, we use *open classes* in addition to the known runtime code evaluation.

> **Open classes** — In a running Ruby application, several application specific objects as well as objects of the core language, like String and Array, exist. The behavior of these objects is completely open to runtime changes.[12] Existing methods can be altered or new methods added. A modification is simple: By re-executing normal Ruby code such as a method declaration, it will override the existing declaration, leading to an immediate modification of the program.

Because of open classes, the already existing classes can be modified to update the behavior of their methods in accordance with the feature configuration.

---

[11] `Proc` objects are immutable when using the standard Ruby library. In [28], however, we explained how to use an external library to produce a string representation of any Ruby object, including `Proc` objects.

[12] This modifiability may create problems when multiple extensions try to override the same method. The Ruby community uses a simple and effective solution: If the method is already defined, then it is aliased to another method name and its behavior is wrapped and preserved inside the new method.

3.4. Usage kinds

Considering the required steps in feature-oriented programming from Section 2.1, rbFeatures implements them as follows (here, a feature means an implementation feature):

- Naming
  - Define conceptual features with their relationships and constraints to other conceptual features (*optional*).
  - Define a product line model by adding all conceptual features (*optional*).
  - Define features with an unique name.
- Identification
  - Identify coarse-grained parts of the application that belong to a feature (modules and classes).
  - Identify fine-grained parts that belong to a feature or even a combination of features (lines or single characters in method bodies).
- Expression
  - Form *feature containment* by defining a *containment condition* and a *containment body*.
- Composition
  - Create a variant from the product line model and instantiate it (*optional*).
  - Activate or deactivate features by accessing the conceptual features of the product line variant (*optional*).
  - Activate or deactivate features by direct manipulation of these entities.
  - Add new variants at runtime (*optional*).

Along these steps, the following usage kinds can be identified.

- Standalone
  - Explicit feature declaration through classes that mix-in the `Feature` module.
  - Features are globally visible entities in the program.
  - Direct manipulation of the feature configuration through a feature's `activate` and `deactivate` method.
- Web Application
  - Implicit feature declaration with central configuration file.
  - Features are visible as both application-specific and application-spanning[13] entities.
  - Direct manipulation of the feature configuration through a feature's `activate` and `deactivate` method. If the feature is application-spanning, then the behavior in the other applications is changed accordingly.
- Product Line Support
  - Declare `FeatureModel` and `ProductLine` entities.
  - Declare `ProductVariant` entities and instantiate them for multiple runtime variants of the application that reside in separated application scopes.
  - Features are defined in the scope of the variants.
  - Manipulation of the feature configuration either through a feature's `activate` and `deactivate` methods or through the `ProductVariant`. In latter case, the configuration is checked against the product line model and only valid configurations change the application's behavior.

As can be seen, rbFeatures supports both standalone applications as well as web applications, and offers optional support with product lines. Using a product line model is recommended though, because otherwise the feature configuration is not validated and could result in a erroneous program.

4. Case study: expression product line

To compare different FOP modularization concepts, the expression problem considers how existing data structures can be extended to represent different types of numbers, expressions and operations. This problem is represented with the Expressions Product Line (EPL) [42]. In terms of rbFeatures' usage kinds, we implemented the EPL as a standalone application with product line support. We first explain some more background, and then walk through the several steps that define the features, the product line model, the implementation, and finally show the usage of variants.

4.1. Background

The EPL is shown as a feature diagram in Fig. 5, which is also called a feature tree because of its nodes that have a definite position. We see that two subfeatures of Numbers exist: Lit (literal) and Neg (negative literal). The numbers can be combined with Add (addition) and Sub (subtraction) Expressions. The Operations added to an expression are Print for printing the textual representation of the expression and Eval for evaluating the expression. As can be seen in Fig. 5, all features except the root feature are optional and the product line can be composed in several ways.

[13] Some web applications are composed of smaller ones, for example the combination of a blog and a wiki. Applications may share similar features such as adding new users. rbFeatures allows one to define such features as application-spanning, which means that its activation and deactivation modifies all individual applications.

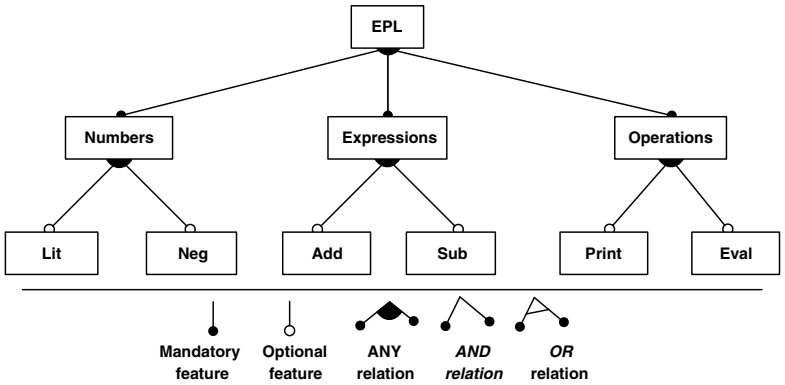

Fig. 5. EPL: feature diagram.

```
1 epl_feature = FeatureModel.configure do
2   name :EPL
3   root
4   subfeatures :Numbers, :Expressions, :Operations
5   requires :EPL => "any :Numbers, :Expressions, :Operations"
6 end
```

Fig. 6. EPL: declaration of the root feature.

```
1 EPL = ProductLine.configure do
2   description "The complete EPL"
3   add_feature epl_feature
4   add_feature numbers_feature
5   #...
6 end
```

Fig. 7. EPL: declaration of the product line.

4.2. Step 1: defining conceptual features and the product line model

The initial step is to provide the conceptual features and a product line model. Therefore, we use a DSL that consists of keywords to express a feature's name, position in the feature tree, its subfeatures, and the constraints it imposes on other features. The declaration starts usually with the root feature and continues down the nodes to the leaves of the feature tree.

Fig. 6 shows how the root feature of the EPL is configured. In Line 1, we use a constructor to define a new feature. The constructor receives a block with several configuration options. Inside the block, Line 2 specifies the name of the feature and Line 3 expresses the position of this feature in the feature tree. Line 4 shows which subfeatures the root has: Numbers, Expressions, and Operations. Finally in Line 5, a constraint is specified. It literally reads, from left to right, that if the EPL feature is activated, then any of its subfeatures can be optionally activated.

After defining the individual features, the next step is to consolidate them in one common product line model. The product line model basically receives all feature model entities and, after checking their validity, allows one to create variant objects out of them. The declaration is straightforward: Additionally to the provision of the features, a `description` is given. Fig. 7 shows an excerpt for declaring the EPL productline. With the features and the product line model in place, we continue to provide the implementation features.

Step 2: feature declaration

Defining features is done with a very concise notation: Classes simply include the `Feature` module. We start to declare the features Numbers, Expressions, and Operations in Fig. 8. As can be seen, they are just classes without additional behavior.

We continue to define features that have additional behavior and exemplify their implementation with Lit, the positive integer, shown in Fig. 9. We use a subclassing relationship to the parent feature `Numbers` to express both the hierarchical relationship of the entities as well as to include the `Feature` module in `Lit` too. The body of the class contains — for the moment — just the `initialize` method, the default method that is executed when a new instance is created. This method stores the passed parameter as the internal integer value.

```
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
```

Fig. 8. EPL: defining the basic features Numbers, Expressions, and Operations.

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

Fig. 9. EPL: defining the feature Lit.

```
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
```

Fig. 10. EPL: defining the feature Print and implementing the `print` method in `Lit` as a feature containment.

Step 3: forming feature containments

After defining `Lit`, `Neg`, `Add`, and `Sub`, we continue with the `Print` feature (cf. Fig. 10, Line 1–2). This feature adds the method `print` to `Lit` — this relationship is expressed as a feature containment in Lines 5–9. Line 5 is the containment condition: Only when the `Print` feature is activated, then the body is executed. The body contains the declaration of the `print` method which outputs the internal value for the `Lit` object.

4.3. Step 4: creating and instantiating variant objects

Finally `Print` and `Eval` are implemented, so that the product line, all conceptual features, and all implementation features are provided. Now we create and instantiate variant objects.

Variant objects receive a unique name and a list of initially activated features. The configuration must be valid according to the product line constraints, only then it can be instantiated. We create a simple variant with the features `Add` and `Lit` (as well as their corresponding parents) and instantiate it in Fig. 11, Lines 1–7. Complementary, a complete variant, in which all features are activated, is created in Lines 9–16.

Step 5: variant usage and runtime modification

In the last step, we use the variants to create some objects and change the feature configuration on the fly. Fig. 12 shows an example for using the interactive Ruby shell. Lines starting with “>>” denote input, and lines with “=>" denote output. In Fig. 12, the following actions occur:

- We begin with creating a single Lit object (Line 1) and a compound Add object (Line 3). Lines 2 and 4 respectively shows the return value of object creation: its in-memory representation. As we see, the name of the variant is available as a scope in the running program. Entities inside the scope are accessed with the name and double colon notation.
- Next we try to use the `print` method of the add expression (Line 6). Since the `Print` feature is not activated, however, the error message `FeatureNotActivatedError` is returned. Such error messages are dynamically defined and inform the user which feature of the containment condition prevents calling the method with the original behavior.

```
1  ProductVariant.configure :name => "SimpleVariant",
2                           :pl => EPL {
3    activate_features :EPL, :Numbers, :Lit,
4                      :Operations, :Add
5  }
6
7  EPL.variant("SimpleVariant").instantiate!
8
9  ProductVariant.configure :name => "ComplexVariant",
10                          :pl => EPL {
11   activate_features :EPL, :Numbers, :Lit, :Neg,
12                     :Operations :Add, :Sub,
13                     :Expressions, :Print, :Eval
14 }
15
16 EPL.variant("ComplexVariant").instantiate!
```

Fig. 11. EPL: Creating and instantiating the `SimpleVariant` and the `ComplexVariant` object.

```
1  >> SimpleVariant::Lit 1
2  => #<Lit:0xb7b35968 @value=1>
3  >> add = SimpleVariant::Add(SimpleVariant::Lit(11), SimpleVariant::Lit(7))
4  => #<Add:0xb7c57544 @right=#<Lit:0xb7c57558 @value=7>, @left#<Lit:0xb7c5756c @value=11>
5
6  >> add.print
7  FeatureNotActivatedError: Feature Print is not activated
8  >> EPL.variant("SimpleVariant").activate_features :Print
9  => :activated
10 >> add.print
11 => 11+7=> nil
12
13 >> add.eval
14 FeatureNotActivatedError: Feature Eval is not activated
15 >>  EPL.variant("SimpleVariant").activate_features :Eval
16 => :activated
17 >> add.eval
18 => 18
19
20 >>  EPL.variant("SimpleVariant").deactivate_features :Print
21 => :deactivated
22 >> add.print
23 FeatureNotActivatedError: Feature Print is not activated
24
25 >> sub = ComplexVariant::Sub(ComplexVariant:Neg(4), ComplexVariant::Lit(2))
26 => #<Sub:0xb7485ac8 @right=#<Lit:0xb74845c4
27  @value=4>, @left#<Lit:0xb74830c0 @value=2>
28 >> sub.eval
29 => -2
```

Fig. 12. EPL: creating variants and modifying their feature configuration.

- To call the `print` method correctly, we use the product line's `variant` method to receive the first-class variant object and activate the `Print` feature in Line 8. Now the method call in Line 10 is successful.
- Using the `Eval` feature is similar, as shown in the Lines 13–18. It needs to be activated first before using it correctly.
- In Lines 20–22 we show how to deactivate features at runtime. The deactivation of PRINT lets the method call to `print` in Line 22 fail accordingly.
- For the complex variant, we create a `Sub` expression in Line 25 and evaluate it in Line 28. This works seamlessly because all features are activated in this variant from the beginning.

4.4. Summary and lessons learned

The current version of rbFeatures is the result of an evolutionary process. The EPL case study accompanied each step. At the very beginning, we tried to use textual annotations in the form of comments. However, we quickly realized that the effort to write a processor for these annotations required one to re-implement bigger parts of Ruby semantics. Also, the granularity of such comments was questionable and the source code would be cluttered.

Then we experimented with Ruby's `Proc` object and the metaprogramming mechanisms to evaluate code at runtime. The concept of feature containments was born. Containments can be built at arbitrary places and granularity, and they are fully integrated in Ruby's semantics. This allowed us to implement features as first-class entities, obtaining a high-level modularization concept. By integrating the DSL for modeling product lines and variants with rbFeatures, we could

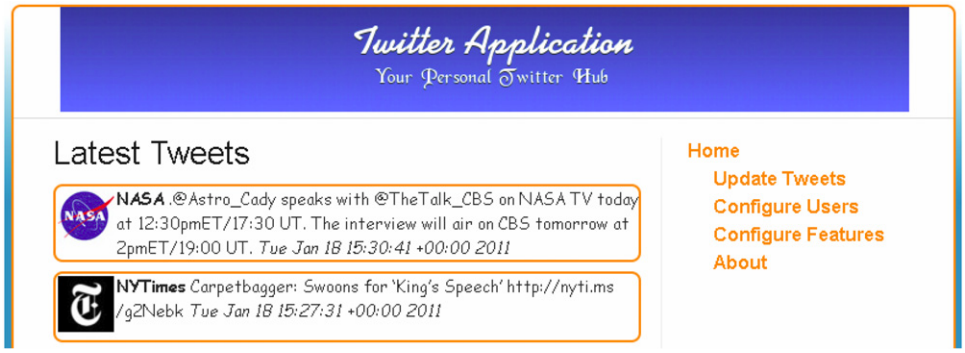

Fig. 13. TAP: index page.

finally close the gap between conceptual and implementation features. An extensive and continuously refined test-case accompanied this process: Tests are specified for different feature granularity, for containment conditions, and for the preservation of variable and method visibility. The EPL provided the background for all those test cases.

Now, the current version of rbFeatures allows one to obtain a complete model of an application. Concrete variants with respect to the product line constraints can be created and modified at runtime. Even the product line model itself can be updated. This flexibility could only be achieved by staying inside the Ruby programming language and building powerful abstractions to represent features.

5. Case study: twitter application

The second case study is a web application in which the feature configuration is accessible via a page. In terms of rbFeatures' usage kinds, this case study is a web application with product line support. We explain how to augment existing web application code with features and how to provide an external configuration for the web application that, among others, specifies the available features and the scope they are residing in.

5.1. TAP — twitter application

The case study is a small application based on the Twitter[14] micro messaging platform. Messages, called tweets in the Twitter jargon, are 140 characters long only. Twitter's compelling feature is the real-time availability of tweets, making it easy to participate live in world-wide discussions about recent topics. TAP can be tried at tap.admantium.com.

Twitter has an API that provides nearly the same functionality as the webpage's user interface: post, delete, and fetch tweets. However, using the official API, the availability of recent tweets is limited: Only the last 200 tweets can be retrieved. The major motivation for TAP is to store tweets for a longer time and thus serve as a repository. TAP has the following functionality:

- Users
  - Add new users
  - Delete existing users
- Tweets
  - View all/latest ten tweets
  - Update tweets for registered users

TAP provides a minimalistic UI shown in Fig. 13. Tweets are shown to the left, and a navigation bar to the right. Two dialogs exist: One for adding or deleting users (Fig. 14), and one for configuring features (Fig. 15). Using rbFeatures, we chose to implement three features: Select and show all tweets in the database (AllTweetsFeature), the addition of new users (AddUserFeature), and the deletion of existing users (DeleteUserFeature).

For implementing web applications, Ruby has a considerable amount of web frameworks to choose from. The most important ones are Rails,[15] Merb,[16] and Sinatra.[17] TAP is implemented in both Rails and Sinatra, but this case study only explains the Sinatra-based application. We choose this variant because Sinatra itself is considered a DSL for web applications, and we use DSL's for database interaction and the HTML/CSS templates as well. Those DSLs are more easy to read and comprehensible for persons familiar with this domain [27].

14 http://twitter.com.
15 http://rubyonrails.org.
16 http://merbivore.com.
17 http://sinatrarb.com.

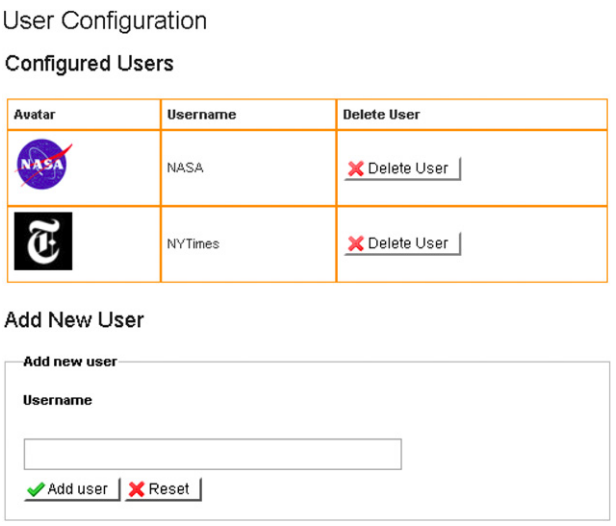

Fig. 14. TAP: user configuration dialog.

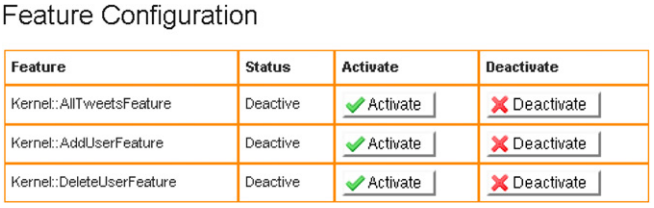

Fig. 15. TAP: feature configuration dialog.

```
1  post '/adduser' do
2    AddUser.code do
3      User.construct params['username']
4      redirect '/tap/user_config'
5    end
6    redirect '/'
7  end
8
9  @@ user_config
10 #...
11 - @add_user_feature.code do
12   %h2 Add New User
13   %form{:action=>'/adduser', :method=>'post'}
14     %fieldset
15       %legend Add new user
16       #...
```

Fig. 16. TAP: relevant source code for the `AddUserFeature`.

The following explanations show how the `AddUserFeature` is implemented. The process to add a new user consists of these steps:

- Parse the request variables and select the submitted username.
- Pass the username to the `User` constructor.
- Query the Twitter API whether there is a user with the specified username.
- When the user exists, then add the user and his latest 200 tweets to the database.
- Redirect the request to the user-configuration dialog.

5.2. Step 1: forming feature containments

The `AddUserFeature` is expressed at two specific places in the application: The *handler* that is called to interact with the Twitter API to add users to the database, and the *template* that renders the user configuration dialog which shows a form for adding new users. The relevant source code for these operations is shown in Fig. 16.

```
1  post '/activate/:feature_name' do
2    feature = FeatureResolver.configured_feature? feature_name
3    feature.activate if feature
4    redirect '/tap/feature_config'
5  end
6
7  post '/deactivate/:feature' do
8    feature = FeatureResolver.configured_feature? feature_name
9    feature.deactivate if feature
10   redirect '/tap/feature_config'
11 end
12
13 @@ feature_config
14 %h1 Feature Configuration
15 %table
16   %thead
17     %td Feature
18     %td Status
19     %td Activate
20     %td Deactivate
21   %tbody
22   - @features.each do |feature|
23     %tr
24       %td= feature.name
25       %td= if feature.active? then 'Active' else 'Deactive' end
26       %td
27         %form{:action=>"/tap/activate/#{feature.name}", :method=>'post'}
28           #...
```

Fig. 17. TAP: extract of the feature modification handlers and the feature configuration dialog.

The handler is shown in Lines 1–7. On Line 1, the `post` defines a HTTP post request handler with the URL `/adduser`. The block contained between the `do...end` keywords is executed when a request with this specific URL is called. Lines 2–5 is a feature containment. The containment condition is simple: Only when the `AddUserFeature` is activated, then the code in Lines 3–4 is executed. This code calls the constructor for a new user by accessing the request parameters for the passed `username` variable. The user constructor interacts with the Twitter API as described above. Afterwards, the request is redirected to render the user configuration dialog. When the `AddUser` feature is not activated, then the request is simply redirected to a handler that renders TAP's root page. These lines show how tight the integration between Sinatra and rbFeatures logic is expressed in the source code.

The template is shown (abbreviated) in Lines 9–16, using a DSL that express HTML tags and their contents. For example Line 12 results in the output `<h2>Add New User</h2>`, and the following lines output a HTML form with buttons to enter and send the new user to be added to TAP. Right in Line 11, we see again how seamless rbFeatures is incorporated into Sinatra: A feature containment begins and completely encompasses the following lines. The form for adding new users is only contained in the HTML output when the `AddUserFeature` is activated.

5.3. Step 2: providing feature configuration support

Web applications are typically hosted on servers provided by a hosting company. Web application owners do not always have direct access to these servers. In order to change the configuration of features, the application itself needs to provide the modification capability. In the case of TAP, our design decision was to add a feature configuration dialog[18] that we showed in Fig. 15.

The feature configuration dialog lists the three features `AllTweetsFeature`, `AddUserFeature`, and `DeleteUserFeature` together with buttons for activating and deactivating them. The buttons themselves trigger request handlers that activate and deactivate the features accordingly. The template and the request handler's implementation is shown in Fig. 17. Inside the handlers, the passed feature name is checked whether it corresponds to a configured feature, and if yes, the respective `activate` or `deactivate` method is called. From Line 13 and onwards, the template code is shown. In Lines 22–27 we see how to use the first-class features to reflect their name and activation status in rendering the template.

5.4. Step 3: external configuration of web applications

While implementing TAP with support for features, we became aware of several challenges of how to integrate rbFeatures with web applications. First, some web applications are actually composed of several smaller ones, such as hosting a blog

[18] For demonstration purpose only, the access to the configuration dialog as well as calling the handlers is open for all users. A non-demonstration application would restrict access to the configuration dialog to privileged users only.

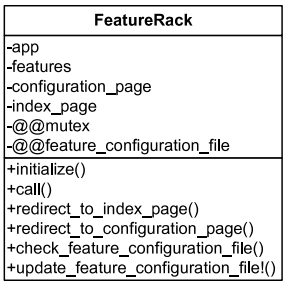

Fig. 18. The `FeatureRack` entity.

application at the "/blog" URL and the TAP application at "/tap". However, rbFeatures normally defines features globally, which could lead to conflicting feature names. We therefore need to define the features in the specific scope of the individual applications. And additionally, we can provide application-spanning features that are globally visible and configure a feature for all singular applications, like adding new users for the blog and TAP. Second, web applications usually consist of several files. Handlers, templates, and auxiliary files can contain code that is part of a feature. All those files must be identified and properly handled by the feature resolver when the feature configuration is modified. And third, web applications are usually multi-threaded to serve several hundred or more requests per minute. We need to find a way of synchronizing the feature configuration between all threads.

These challenges do not influence how rbFeatures works, but require adapting the startup process for the web application and the feature initialization. We decided to master these challenges by extending the configuration file which every Ruby web application needs to define.

In Ruby, all web application frameworks are compatible with Rack.[19] Rack (a) provides a common data format for request and response objects that are produced by the web applications, (b) allows passing these object to other web applications that serve as filters, and (c) implements a simple configuration language to compose several single web applications with different base URLs into one compound application.

Rack governs the web application's configuration and startup process. We wrote a small Rack extension, called `FeatureRack`, which adds more configuration options and incorporates rbFeatures and the feature declaration into the web applications startup process. `FeatureRack`, shown in Fig. 18, adds two important configuration options to hosting web applications: The ability to specify a list of application-specific or application-spanning features (which are declared in the respective scope when the application is loaded) and to specify the source code files that include feature containments (the content of these files is added as a string representation to the `FeatureResolver`).

Let us consider the tasks of `FeatureRack` in the context of hosting TAP together with a blog application. In Fig. 19, the blog application runs at the "/blog" URL (Lines 1–8), and the TAP application runs at the "/tap" URL (Lines 10–18). While the shown expressions `map`, `use`, and `run` are already provided by Rack, other expressions are part of `FeatureRack` and help to address the following points:

- *Define application-specific features* — Lines 3 and 12 define features specific for the scope of the respective application. Inside the scope `BlogApp`, the features `AddEntryFeature` and `DeleteEntryFeature` are configured, and in the `Tap` scope, the features `DeleteUserFeature` and `AllTweetsFeature` are defined.
- *Define application-spanning features* — Lines 4 and 12 create the global `AddUserFeature` in the `Kernel` scope, which is global in the Ruby interpreter process.
- *Configure multiple files* — Complex web applications may consist of several files that define helpers, business logic, handlers, and templates. All files that include feature containments must be added to rbFeatures `FeatureResolver` to be re-evaluated when the feature configuration changes. In Lines 5 and 14 these files are configured.
- *Configure index page* — Lines 6 and 15 configure a default index page that is shown if a request fails.
- *Feature configuration page* — The `FeatureRack` adds a default feature dialog showing the current feature configuration. Some applications, such as TAP, provide a custom dialog, which is expressed in Line 16 and overrides the default dialog.

To solve the challenge of synchronizing the feature configuration between multiple threads of the application, `FeatureRack` uses a shared resource, the feature configuration file, that contains the current feature configuration.

Changing the feature configuration starts with a user operating the feature diagram editor. Upon a change, a global mutex,[20] which is shared between all application threads, is used to access the configuration file. The mutex synchronizes all threads for gaining the sole access to the configuration file. Immediately the file is locked, then the new configuration is written to the file, and the file is unlocked again. If there should be any other attempt to change the feature configuration

19 http://rack.rubyforge.org/.

20 To facilitate concurrent programming, Ruby supports mutexes as first-class objects.

```
map '/blog' do
  use FeatureRack,
    BlogApp => [:AddEntryFeature, :DeleteEntryFeature],
    Kernel => [:AddUserFeature],
    :files => [File.join(Dir.pwd,"lib/blog_app.rb")],
    :index_page => '/blog'
  run BlogApp
end

map '/tap' do
  use FeatureRack,
    Tap => [:DeleteUserFeature, :AllTweetsFeature],
    Kernel => [:AddUserFeature],
    :files => [File.join(Dir.pwd,"lib/tap.rb")],
    :index_page => '/tap'
    :configuration_page => '/tap/feature_config'
  run Tap
end
```

Fig. 19. TAP: external configuration file for the web application.

meanwhile, this particular request will be buffered until the file is unlocked again because of the mutex. These steps, however, only modify the feature configuration in the particular thread that processed the request — other threads still have the old configuration.

To update all application threads with the new feature configuration, we implemented a "lazy update" strategy. Each thread periodically checks the timestamp of the configuration file. Upon noticing a changed timestamp, the thread will first serve all buffered request, then update its feature configuration, and then start to serve all requests that were received in the meanwhile. The stored requests and all following ones operate with the new configuration. This strategy successfully prevents the complete blocking of the application because some threads still receive and serve requests, and no inconsistent requests are served because each thread cleanly separates request serving before and after changing its feature configuration.

5.5. Lessons learned and summary

In our first experiment with using web applications and rbFeatures, we were surprised — because rbFeatures worked out of the box. We simply added classes representing features to the application, added request handler that could change the feature configuration, and loaded the complete application code as a string with the `FeatureResolver`. These steps were straightforward. We were surprised because of the ease how rbFeatures' expressions integrate with the other DSLs that TAP uses (for HTML rendering, for database transactions, for handler declaration), and that we did not need to change any rbFeatures semantics.

After this inception, however, we realized that there are particular challenges with web applications that we need to regard. Web applications can be composed of other web applications, and this composition allows features to be specific for a single application or to be global for all application in the composition. And web applications are usually multi-threaded, so we need a way to synchronize the feature configuration between all threads. These challenges were solved by implementing `FeatureRack`, an extension of Rack, the underlying framework for all Ruby web applications. `FeatureRack` governs the configuration of application-specific of application-spanning features, initializes the web applications by handling their source code to the `FeatureResolver`, adds request handlers that can be used to change the feature configuration, and synchronizes the feature configuration between the threads.

In summary, the basic rbFeatures implementation proved to be very robust and is cleanly separated from the application logic — it can interact with other Ruby DSLs down to integrating several DSL expressions. And our experiences with web applications showed that writing small extensions to cover specific requirements is the best approach to bring rbFeatures to other application domains as well.

6. Related work

There are several approaches to feature modeling (providing conceptual features) and feature implementation (providing concrete features). Because the feature implementation part defines how well features in the sense of first-class entities are integrated with the program, we focus on these approaches and only give a small introduction to feature modeling approaches.

6.1. Feature modeling approaches

Approaches to feature modeling can be differentiated into graphical ones that come with an editor support and offer various visualizations, and textual ones that use dedicated languages to model features and their constraints.

6.1.1. Graphical approaches

Graphical approaches are CaptainFeature,[21] CBFM[22] (cardinality-based feature modeling) [23], COVAMOF (Configuration in Industrial Product Families Variability Modeling Framework) [50], pure::variants[23] [15], and FeatureIDE [39]. CaptainFeature allows one to model feature diagrams with a cardinality-based approach.

CBFM allows one to model feature trees in the form of lists. Symbols show which features are currently selected and which ones cannot be selected due to constraints. COVAMOF supports dedicated view for modeling variation points and dependencies between features [50]. pure::variants allows users to model a feature-tree graphically and to compare variants in a matrix that represents the feature's inclusion/exclusion [15]. FeatureIDE is an open source projects that combines graphical editor for feature models with text editors for specific feature implementation approaches like AHEAD or FeatureHouse [39].

6.1.2. Textual approaches

Textual approaches are FDL (Feature Description Language) [56], VSL (Variability Specification Language) [12], AMPL (Asset Model for Product Lines) modeling language [50], Guidsl [10], and TVL (Text-based Variability Language) [17]. The language representation of conceptual features in these approaches is based on the premises that (1) a language representation is amenable to automatic processing of various kinds, (2) a language representation enables managing variability in a uniform manner, and (3) equivalent textual notations in the form of dedicated language constructs enable scalability for feature models as well as to make them understandable to most stakeholders.

For example, FDL allows one to express constraints as mandatory or optional relationships between features and also user preferences can be modeled [56]. Guidsl uses attribute grammars for modeling rich relationships, and it additionally generates a GUI in which a feature configuration can be created interactively [10]. While these languages only support conceptual features, there are two languages that support references from conceptual features to architectural components: the Variability Modeling Language (VML) [44] and Koalish [8]. VML allows features to express the modifications of the component, interaction, and deployment of an architecture's elements. A feature configuration triggers the specified actions that can connect, add, remove, or deploy those architectural elements [44].

6.1.3. Discussion

Feature modeling approaches mainly focus on conceptual features. They treat the creation of concrete product variants only as composition of a program's components. Only recently, Clafer, which is still in development, proposes a textual language that models feature diagrams and allows one to reference implementation entities of a class diagram such as classes and their fields [16]. If elaborated, such a representation could provide a similar unified notation of conceptual and concrete features as in rbFeatures.

The study [51] discusses most of the mentioned approaches in more detail, including properties how the modeling perspective can be extended to the implementation perspective by forming suitable abstraction levels that represent the program's components. Other studies compare feature models and what kind of automated analyses such as determining the feature model satisfiability or finding dead features can be done [24,13].

rbFeatures combines feature modeling and implementation. Constraints such as mandatory, optional, and alternative features are expressible. The product line model is available to the running program and can be used for reflection and modification. For concrete features, rbFeatures uses a powerful abstraction to express feature containments in a program's implementation. Because coarse- and fine-grained parts are expressible, there is no need for additional language constructs such as in VML. The language that is used to express the containments is as powerful as the program's implementation language: they are the same. First-class features facilitate expressing the program's behavior and the semantics of feature composition side-by-side.

6.2. Feature implementation approaches

There is a great variety of feature implementation approaches. Approaches can be dependent or independent of a program's programming language and they can use separate feature artifacts or annotations [2]. We separate the approaches into three distinct types: annotation-based, modularization-based, and refinement-based. Each of them differs how features are represented and composed, which is detailed in the following sections.

6.2.1. Annotation-based representations

One type of annotations are *source code annotations* such as "#ifdef" statements in C++. The code belonging to different features is simply put inside preprocessor directives. The preprocessor removes those annotations that do not belong to the selected feature configuration, and the remaining code is compiled [35]. One example for annotation-based representations

21 http://captainfeature.sourceforge.net/.

22 http://sourceforge.net/projects/fmp/.

23 http://www.pure-systems.com/Details.53.0.html.

is the tag and prune approach [18]. This approach adds tags inside C source code that express whether blocks of code should remain inside the source code. A self-written tool checks the source code and removes all parts that are not part of the specified feature configuration.

Another kind are *virtual annotations*. Instead of modifying the source code directly, annotations are added with the help of a tool upon a suitable program representation such as an abstract syntax tree [38]. Using this tool to configure a variant, the program is then composed from its virtually annotated parts. An example for this technique is XVCL, an approach that supports XML-based annotations for a wide variety of program artifacts, including documentation and source code [60].

Representation

In these approaches, features are not concrete entities but consist of annotations of source code parts. These annotations are either placed directly in the source code in the form of comments, or virtually on top of the program or a suitable program representation (like its abstract syntax tree). However, representations for the conceptual features, the product line, or the product variants — if they exist — are defined in the scope of the tools that process the annotations. For example information about the features and their valid combinations to form a variant are expressed as configuration dialogs [38]. In contrast, rbFeatures combines the representation of the product line model, feature constraints, and first-class features to become parts of the application.

Composition

The composition combines the base program with those parts that belong to features in the specified feature configuration. Compiling this composition yields the specific variant. Because the variant is compiled, the information about other possible variants is removed from the code and the variant cannot be changed afterwards. rbFeatures retains this information and allows even runtime adaptation of the product lines and the variants (see [29]).

6.2.2. Modularization-based approaches

These approaches use the programming language given modularization concepts to represent implementation features. They implement the variation that a particular feature configuration exerts as the application of source code fragments introduced by the modularization concepts. Among these approaches are traits in Scala, hyperslices in HyperJ, atoms and units in Jiazzi (see [43] for an explanation and comparison), Classboxes [14], CaesarJ [7], and Object Teams/Java [31].

Representation

These approaches model features in terms of the given modularization concept. Features are not explicitly contained in the approaches, but indirectly expressed via the application of the modularization concepts. Likewise representations for product lines and variants do not exist. rbFeatures adds features as entities to the program that can be tightly integrated with the application's logic, as well as providing representations for software product lines and variants.

Composition

Using the existing modularization concepts determines what and how functionality can be added or removed to a program. We want to give some examples for the available approaches.

Traits in Scala are a modular representation of reusable behavioral concerns [49]. While they allow to add even generic functions, traits cannot be removed from the classes [58], and thus once composed behavior can not be removed from the application. Hyperslices group all program units implementing a particular feature [43], but once the program is composed, features cannot be traced and it is difficult to comprehend how the program is affected when new features are added [19]. rbFeatures uses the modularization capabilities of `Proc` objects to form the feature containments. Ruby's interpreted nature enables runtime adaptation of the program by executing these objects and modifying the program's behavior. Classboxes support class extensions that are visible locally within the namespace the classbox provides [14]. This behavior impacts how to structure the whole application resulting in more extensive feature refactorings then those required by rbFeatures.

6.2.3. Refinement-based approaches

The last type of approaches separate a feature-based program into a base program and refinements that add specialized behavior. Products are created by composing the base program with the refinements that represent concrete feature. Typically, these approaches add specific language constructs to express these refinements.

Refinements can take several forms. Some approaches use the keyword `refine` as a language construct (AHEAD tool suite [9], FeatureC++ [5], FeatureHouse [3], and Fuji [4]). Other approaches introduce concepts similar to refinements, such as aspects from aspect-oriented programming [41] or contexts from context-oriented programming [32]. Aspects define behavior which cross-cuts several parts of a program. Implementations like AspectJ [36] introduce aspects that can be used to represent features [1]. Contexts are concepts that represent a particular behavior of the application and are represented using first-class entities called layers [32]. Contexts can be seen as features, and composing the contexts is similar to providing a feature configuration. Implementations for context-oriented programming are available for Java, Ruby, Python and other languages (see [6] for an explanation and comparison of these implementations).

Representation

The approaches that use `refinement` as a language keyword treat features as refinements to base classes. Refinements are stored in folders and the relations between features are represented with a folder hierarchy. Features are combined with the base program to create a concrete variant. Although some parts of the features, like their name, are added explicitly to the language, their implementation is still external to the program. Also, the language processor needs to be extended to support these keywords. For example in Fuji, the modular compiler JastAddJ is used for this extension [4]. Again, no representations for conceptual features, product lines and variants can be found in these approaches.

rbFeatures uses an internal representation for the features. Developers can see the behavior of features when they work on the source code. Representations for software product lines and variants are also included in rbFeatures.

Aspects can help to represent static and dynamic features (modifiable at runtime) [1] at coarse- and fine-grained levels. Different tools treat aspects as external or internal to the program. AspectJ defines aspects as external entities that identify a join point and express the advice that is executed when this join point is reached in the program's execution. The keywords for these abstractions require an external tool, the aspect weaver, to work [36], while rbFeatures uses Ruby semantics. Classpects is an advanced aspect-oriented programming approach that integrates object-oriented and aspect-oriented expressions, such as adding aspect code directly in the declaration of a class [47]. Both this integration and the introduction of aspects as first-class entities are similar to rbFeatures. However, the nature of aspects is to modify the program in join points that are identified at compilation or runtime. When the developer works on a particular piece of code, he has no information if an aspect modifies this piece too, and consequently may introduce errors. In rbFeatures, the representation of code belonging to feature and common program code is unified.

There is a strong similarity between rbFeatures and context-oriented programming. Features and layers are first-class entities of the program. Both can be activated and deactivated at runtime, immediately introducing new behavior. The difference lies in the conceptualization of these behavioral changes. Features represent core functionality of a program that can be composed independently of each other (with respect to constraints). Layers represent behavioral adaptation dependent on the program's actors (entities that interact with the program) and the program's environment [32]. Such behavioral adaptation has not the nature of activating or deactivating a single functionality, but to change several functions of the program. Therefore, we see the typical role of layers as providing a adapted program variant. However, introducing the concepts of atomic and composite layers as shown in [21] can lead to regarding layers and features as conceptually very similar entities.

Composition

In the AHEAD tool suite and its similar approaches as given above, composition is carried out using a textual description of features that are called equation files. The equation files treat the folder hierarchy as a feature model and they include composition constraints. AHEAD allows to check the safe composition of refinements and the base classes by using propositional formulas that express the relations between features. rbFeatures uses the product line model to check a particular feature configuration to the rules and constraints that have to be regarded. This model can be even changed at runtime. Ruby is a dynamically typed language, so no type checks are available. However, as we proposed earlier, Ruby has powerful testing frameworks that can be used to define behavioral tests. Writing the tests in such a way that they are testing the features allows implementing the application with high confidentially in its correct behavior.

A related approach to the way how variants are treated in rbFeatures is presented in the SPL-API, an extension of FeatureC++ [48]. In this approach, a method receives a list of features, compares the configuration with the feature model (expressed in an XML file), and uses the FeatureC++ code base to compose the variant. However, the feature-model is not expressed as first-class objects and it is not explained whether runtime modification of the feature model and the variants are supported.

Aspects and layers differ in their composition mechanisms. Aspect-oriented programming approaches use the aspect weaver to compose the base program and the aspects. Aspects define certain execution points of the program, called join points, where the aspect is composed. However, common implementations have a restricted set of join point at method execution and field access level [40] while rbFeatures' containments can be placed anywhere in the source code (in this way, feature containments can be seen as join points that can be placed arbitrarily.) Context-oriented programming uses techniques such as partial declaration of modules, classes, and functions in a layer that modifies the behavior upon layer activation. Layers are first-class entities that can be used and modified throughout the program. As explained before, we see a similarity between layers and rbFeatures' notation of features, and the used abstractions are similar powerful in effectuating behavioral modifications. However, rnFeatures' use of the product line model to explicitly create variants with different feature configurations at runtime is not found in context-oriented approaches.

6.3. Summary

rbFeatures is a hybrid approach that mixes the strength of several singular approaches. First, it is an annotation-based approach that adds source code annotations. However, these containment are not comments — they use an existing construct of Ruby, the `Proc` objects. `Proc` objects can be used for coarse-grained and fine-grained containments for flexible composition, and because rbFeatures reuses these abstractions, is is also an modularization-based approach. Finally, features

and containments are explicitly represented in the programs and add certain keywords to the Ruby language, which is similar to refinement-based approaches.

Yet rbFeatures is unique in several ways. First, it provides several implementation benefits. rbFeatures is defined entirely on top of the abstractions which the Ruby programming language offers: the composition capabilities of `Proc` objects and the support for metaprogramming including runtime-code evaluation, hooks, and open classes. Therefore, rbFeatures can be considered a DSL for FOP that adds language constructs for providing features as high-level modularization concepts. When developers use rbFeatures, they see the behavior which is introduced with a certain feature while they implement the program — they do not need to study an external representation but put code in containments. rbFeatures runs on all Ruby interpreters that conform to the core language specification. Second, it provides benefits in terms of representing and composing conceptual and implementation features. There is no fixed composition order of features, and behavioral changes to a variant that were introduced with a feature can be reverted. rbFeatures explicitly expresses features, product lines, and variants as first-class entities. There is no need to form external representations of constraints or rules. The semantics of features can be tightly integrated with a program's behavior, as we showed for the TAP case study (see [27]). Different variants with a unique feature configuration can be created and added to the program. Product lines and the variants can be modified at runtime, providing powerful runtime adaptation capabilities for programs [29]. Comparing to other approaches, there is no gap between modeling and implementing features — they happen at the same abstraction level and are available to the program at runtime.

7. Summary

rbFeatures is a pure language extension to Ruby to enable feature-oriented programming. We introduced the Ruby language, and presented a step-by-step walkthrough how rbFeatures works internally. By reporting the Expressions Product Line case study and a web application, we saw how to use rbFeatures in different scenarios and different programs.

Defining features as first-class entities leads to abstractions that can be used anywhere in the program. The logic of feature composition that depends on one or several activated features is seamlessly integrated with the program. Features can be stored in variables, returned from methods and extended like any other object. Developers can use these abstractions to obtain a high-level modularization concepts that facilitates program customization. When working on the source code, developers see the feature containments and can understand the different behavior of the program based on the feature configuration. Our experience in feature-refactoring existing applications showed that often only minimal changes to the program are necessary. Finally, because rbFeatures is a extension to Ruby that only uses the existing abstractions, rbFeatures is useable with all virtual machines that implement the complete Ruby language specification.

In addition to standalone applications, rbFeatures also masters the specific challenges of web applications. Developers can focus on using feature containment inside their web application only. The `FeatureRack` extension provide requests handlers that enable feature configuration changes, supports compound web applications consisting of single web applications, and defines application-specific and global features. Any Ruby Rack-compatible web framework can use rbFeatures now.

In future research, we want to extend the representation capabilities of rbFeatures.

Acknowledgements

We thank Christian Kästner, Sven Apel, Marco Fischer and the anonymous reviewers for comments on an earlier draft of this paper.

References

[1] S. Apel, The role of features and aspects in software development, Ph.D. Thesis, Otto-von-Guericke-Universität Magdeburg, Germany, 2007.
[2] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology (JOT) 8 (5) (2009) 49–84.
[3] S. Apel, C. Kästner, C. Lengauer, FeatureHouse: language-independent, automated software composition, in: Proceedings of the 31st International Conference on Software Engineering, ICSE, IEEE Computer Society, Washington, 2009, pp. 221–231.
[4] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, T. Leich, Access control in feature-oriented programming, Science of Computer Programming 77 (3) (2012) 174–187.
[5] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming, in: R. Glück, M. Lowry (Eds.), 4th International Conference on Generative Programming and Components Engineering, GPCE, in: Lecture Notes in Computer Science, vol. 3676, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 125–140.
[6] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A comparison of context-oriented programming languages, in: International Workshop on Context-Oriented Programming, ACM, New York, 2009, pp. 1–6.
[7] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An overview of CaesarJ, in: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development I, in: Lecture Notes in Computer Science, vol. 3880, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 135–173.
[8] T. Asikainen, T. Soininen, T. Männistö, A Koala-based approach for modelling and deploying configurable software product families, in: F. van der Linden (Ed.), 5th International Workshop on Software Product-Family Engineering, PFE, Springer-Verlag, Berlin, Heidelberg, 2004, pp. 225–249.
[9] D. Batory, Feature-oriented programming and the AHEAD tool suite, in: Proceedings of the 26th International Conference on Software Engineering, ICSE, IEEE Computer Society, Washington, 2004, pp. 702–703.
[10] D. Batory, Feature models, grammars, and propositional formulas, in: H. Obbink, K. Pohl (Eds.), Proceedings of the 9th International Conference on Software Product Lines, SPLC, in: Lecture Notes in Computer Science, vol. 3714, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 7–20.

[11] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering, ICSE, IEEE Computer Society, Washington, 2003, pp. 187–197.
[12] M. Becker, Towards a general model of variability in product families, in: Software Variability Management Workshop, Groningen, Netherlands, Feb. 2003, pp. 19–27.
[13] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: a literature review, Information Systems 35 (6) (2010) 615–636.
[14] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Classboxes: controlling visibility of class extensions, Computer Languages, Systems & Structures 31 (3) (2005) 107–126.
[15] D. Beuche, H. Papajewski, W. Schröder-Preikschat, Variability management with feature models, Science of Computer Programming 53 (3) (2004) 333–352.
[16] K. Bąk, C. K., A. Wąsowski, Feature and class models in clafer: Mixed, specialized, and coupled, in: 3rd International Conference on Software Language Engineering, ICSE, Post proceedings, 2010 (in press).
[17] Q. Boucher, A. Classen, P. Faber, P. Heymans, Introducing TVL, a text-based feature modelling language, in: Benavides, D., Batory, D., Grünbacher, P. (Eds.), Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems, VAMOS, No. 37 in ICB Research Reports, University of Duisburg-Essen, Germany, 2010, pp. 159–162.
[18] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, L. Demonceau, Tag and prune: a pragmatic approach to software product line implementation, in: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, ASE, ACM, New York, 2010, pp. 333–336.
[19] R. Chitchyan, I. Sommerville, A. Rashid, A model for dynamic hyperspaces, in: Workshop on Software Engineering Properties of Languages for Aspect Technologies, SPLAT, 2003. Available online http://aosd.net/workshops/splat/2003/–papers/Ruzanna_Chitchyan.pdf.
[20] J.O. Coplien, Multi-paradigm design, Ph.D. Thesis, Vrije Universiteit Brussel, 2000.
[21] P. Costanza, T. D'Hondt, Feature descriptions for context-oriented programming, in: Thiel, S., Pohl, K. (Eds.), Proceedings 12th International Conference for Software Product Lines, SPLCL, 2nd International Workshop on Dynamic Software Product Lines, DSPL, Vol. 2 (Workshops), University of Limerick, Ireland, 2008, pp. 9–14.
[22] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Boston, San Franciso et al., 2000.
[23] K. Czarnecki, S. Helsen, U.W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (1) (2005) 7–29.
[24] David Benavides, P.T. Antonio Ruiz-Cortés, S. Segurae, A survey on the automated analyses of feature models, in: J.C.R. Santos, P. Botella (Eds.), Proceedings of the 11th Jornadas de Ingenier del Software y Bases de Datos, JIBSD, 2006, pp. 367–376.
[25] D. Flanagan, Y. Matsumoto, The Ruby Programming Language, O-Reilly Media, Sebastopol, USA, 2008.
[26] S. Günther, Engineering domain-specific languages with ruby, in: H.-K. Arndt, H. Krcmar (Eds.), 3. Workshop des Centers for Very Large Business Applications, CVLBA, Shaker, Aachen, 2009, pp. 11–21.
[27] S. Günther, Multi-DSL applications with Ruby, IEEE Software 27 (5) (2010) 25–30.
[28] S. Günther, S. Sunkle, Enabling feature-oriented programming in Ruby, Technical Report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, Germany, 2009.
[29] S. Günther, S. Sunkle, Dynamically adaptable software product lines using Ruby metaprogramming, in: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD, ACM, New York, 2010, pp. 80–87.
[30] J. Heering, Application software, domain-specific languages, and language design assistants, Technical Report sen-r0010, Center for Mathematic and Computer Science, University of Amsterdam, 2000.
[31] S. Herrmann, A precise model for contextual roles: the programming language ObjectTeams/Java, Applied Ontology 2 (2) (2007) 181–207.
[32] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151.
[33] P. Hudak, Modular domain specific languages and tools, in: P. Davenbu, J. Poulin (Eds.), Proceedings of the Fifth International Conference on Software Reuse, ICSR, IEEE, Washington, 1998, pp. 134–142.
[34] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-Oriented Domain Analysis (FODA) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, USA, 1990.
[35] C. Kästner, S. Apel, Virtual separation of concerns — a second chance for preprocessors, Journal of Object Technology (JOT) 8 (6) (2009) 59–78.
[36] C. Kästner, S. Apel, D. Batory, A case study implementing features using AspectJ, in: Proceedings of the 11th International Software Product Line Conference, SPLC, IEEE Computer Society, Washington, 2007, pp. 223–232.
[37] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the 30th International Conference on Software Engineering, ICSE, ACM, New York, 2008, pp. 311–320.
[38] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, D. Batory, Guaranteeing syntactic correctness for all product line variants: a language-independent approach, in: M. Oriol, B. Meyer (Eds.), Proceedings of the 47th International Conference on Objects, Components, Models and Patterns (TOOLS Europe), in: Lecture Notes in Business Information Processing, vol. 33, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 175–194.
[39] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, S. Apel, FeatureIDE: a tool framework for feature-oriented software development, in: Proceedings of the 31st International Conference on Software Engineering, ICSE, IEEE Computer Society, Washington, 2009, pp. 611–614.
[40] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J.L. Knudsen (Ed.), Proceedings of the 15th International Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 2072, Springer-Verlag, Berlin, Heidelberg, 2001, pp. 327–354.
[41] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit, S. Matsuoka (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Berlin, Heidelberg, 1997, pp. 220–242.
[42] R.E. Lopez-Herrejon, D. Batory, A standard problem for evaluating productline methodologies, in: J. Bosch (Ed.), Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering, GCSE, in: Lecture Notes in Computer Science, vol. 2186, Springer-Verlag, London, 2001, pp. 10–24.
[43] R.E. Lopez-Herrejon, D. Batory, W. Cook, Evaluating support for features in advanced modularization techniques, in: A.P. Black (Ed.), Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 3586, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 169–194.
[44] N. Loughran, P. Sánchez, A. Garcia, L. Fuentes, Language support for managing variability in architectural models, in: C. Pautasso, E. Tanter (Eds.), Proceedings of the 7th International Symposium on Software Composition, SC, in: Lecture Notes in Computer Science, vol. 4954, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 36–51.
[45] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, in: R.N. Taylor, M.B. Dwyer (Eds.), Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, ACM, New Yok, 2004, pp. 127–136.
[46] C. Prehofer, Feature-oriented programming: a fresh look at objects, in: M. Aksit, S. Matsuoka (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Berlin, Heidelberg, 1997, pp. 419–443.
[47] H. Rajan, K.J. Sullivan, Classpects: unifying aspect- and object-oriented language design, in: Proceedings of the 27th International Conference on Software Engineering, ICSE, ACM, New York, 2005, pp. 59–68.
[48] M. Rosenmüller, N. Siegmund, G. Saake, S. Apel, Code generation to support static and dynamic composition of software product lines, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE, ACM, New York, 2008, pp. 3–12.
[49] N. Scharli, S. Ducasse, O. Nierstrasz, A. Black, Traits: composable units of behaviour, in: Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP, Vol. 2743, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 327–339.

[50] M. Sinnema, O. De Graaf, J. Bosch, Tool Support for COVAMOF, in: Proceedings of the 2nd Workshop on Software Variability Management for Product Derivation, SVM, 2004. Available online http://www.msinnema.nl/SVMWorkshopMocca.pdf.
[51] M. Sinnema, S. Deelstra, Classifying variability modeling techniques, Information & Software Technology 49 (7) (2007) 717–739.
[52] Y. Smaragdakis, 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) (2002) 215–255.
[53] S. Sunkle, M. Rosenmüller, N. Siegmund, S.S. Rahman, G. Saake, S. Apel, Features as first-class entities - toward a better representation of features, in: Loughran, N., Groher, I., Lopez-Herrejon, R., Apel, S., Schwanninger, C. (Eds.), Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, McGPLE, Technical Report, Number MIP-0804, Department of Informatics and Mathematics, University of Passau, Germany, 2008, pp. 27–34.
[54] P. Tarr, H. Ossher, W. Harrison, S.M.J. Sutton, N degrees of separation: multi-dimensional separation of concerns, in: Proceedings of the International Conference on Software Engineering, ICSE, ACM, New York, 1999, pp. 107–119.
[55] D. Thomas, C. Fowler, A. Hunt, Programming Ruby 1.9 - The Pragmatic Programmers' Guide, The Pragmatic Bookshelf, Raleigh, USA, 2009.
[56] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, Journal of Computing and Information Technology 10 (1) (2002) 1–18.
[57] A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, ACM SIGPLAN Notices 35 (6) (2000) 26–36.
[58] D. Wampler, A. Payne, Programming Scala, O'Reilly Media, Sebastopol, USA, 2009.
[59] J. Withey, Investment analysis of software assets for product lines, Technical Report CMU/SEI96-TR-010, Software Engineering Institute, Carnegie Mellon University, USA, 1996.
[60] H. Zhang, S. Jarzabek, XVCL: a mechanism for handling variants in software product lines, Science of Computer Programming 53 (3) (2004) 381–407.