Technical Report

Nr.: FIN-15-2008

Refactoring Feature Modules

Martin Kuhlemann

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-15-2008

Refactoring Feature Modules

Martin Kuhlemann

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 10 MDStV):

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

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Kuhlemann
Postfach 4120
39016 Magdeburg
E-Mail: martin.kuhlemann@ovgu.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 52

Redaktionsschluss: 18.12.2008

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Refactoring Feature Modules

Martin Kuhlemann
University of Magdeburg, Germany
martin.kuhlemann@ovgu.de

Don Batory
University of Texas at Austin, USA
batory@cs.utexas.edu

Sven Apel
University of Passau, Germany
apel@uni-passau.de

Abstract

Feature-oriented design improves reusability of object-oriented classes. Features are increments in program functionality and correspond to feature modules. Programs composed from feature modules can conflict to applications they interact with. In this paper, we introduce *refactoring feature modules*. Refactoring feature modules are a new combination of feature-oriented design and refactorings that adjust members and classes that are composed from feature modules. We show that refactoring feature modules reduce conflicts between applications and encourage software reuse.

1. Introduction

Feature-oriented design extends object-oriented design to improve reusability of object-oriented classes [43, 3]. Features are increments in program functionality and correspond to feature modules. Feature modules add classes to a program, members to classes, or extend members of classes.

Features are organized in a feature model that defines valid compositions of features. Stakeholders use the feature model to derive different programs but have no knowledge about the program's implementation. If the program, that is composed from feature modules, is a library (so-called *scaling software library* [4]) then the library interacts with classes that are not part of the feature-oriented design. However, library classes and members may preclude the library's integration into another software [35, 20, 22, 39], e.g., if library members are expected to have a different name. In that case of conflicts, the stakeholder must change the library she composed and so she must learn the library's implementation first.

An object-oriented refactoring describes a process to restructure and change object-oriented members and classes [16]. A refactoring may remove conflicts caused by classes and members and, thus, facilitate their reuse. But, even if a refactoring possibly performed on every program is known upfront, existing refactoring approaches are not integrated into the feature-oriented configuration process. Instead, refactorings (or according meta-programs) must be executed after the program is composed. However, the developer cannot compose every valid program in advance [25]. To refactor multiple programs efficiently, refactorings must be integrated into the feature-oriented configuration process and feature-oriented design.

In this paper, we propose to define refactorings in refactoring units and encapsulate these refactoring units in feature modules of feature-oriented design, we call feature modules that encapsulate refactoring units *refactoring feature modules*. We show how refactoring units in a feature-oriented design benefit from feature modules and show how the programs composed from feature modules benefit from refactoring feature modules. First, we show how refactoring feature modules reduce conflicts that precluded reuse of libraries in the past and discuss the role of refactoring feature modules in other areas where feature-oriented design is used. Second, we show how refactoring units can be composed with feature-oriented mechanisms.

Relationship of refactorings and feature modules. Refactorings and feature modules are not competing concepts but expose interesting complements. Refactorings and feature modules both are transformations executed on programs. Thereby, a refactoring is a *process* of transforming the members and classes of a program (so-called *system architecture* [18]) whereas feature modules are *language constructs* that transform a program's *functionality*. To transform a program, refactorings restructure existing members and

```
1  public class Deque{
2   List _elements;
3   void insert_front(Element e){
4    _elements.add(e);
5   }
6  }
```

(a) Jak module *L1*

```
7  refines class Deque {
8   int _depth;
9   void setElements(List newElems){
10   _elements=newElems;
11  }
12  void insert_front(Element e){
13   Super.insert_front(e);
14   _depth=_elements.size();
15  }
16 }
```

(b) Jak module *L2*

```
17 public class Container {
18  Deque _d;
19  void add_front(Element e){
20   _d.insert_front(e);
21  }
22  void setElements(List newElems){
23   _d.setElements(newElems);
24  }
25 }
```

(c) Jak module *L3*

Figure 1. Sample feature-oriented design from [4].

classes, whereas feature modules create new members and classes or extend them. Feature modules transform methods and classes individually (so-called heterogeneous crosscutting [1]), whereas one refactoring oftentimes transforms different members and classes at once (so-called homogeneous crosscutting [1]). Feature modules are abstracted with a name; refactorings are not defined in modules and, thus, there is no abstraction for refactorings. Consequently, to execute a feature module on members and classes one only has to know the according feature module's name but to refactor a program one has to know the program's system architecture and the refactoring effects.

2. Background

In this section, we review concepts of feature-oriented design and refactorings.

2.1 Feature-oriented Design

A feature is an increment in program functionality which in feature-oriented design is implemented by a feature module. A feature module encapsulates classes or *class refinements* [3]. Class refinements add members to classes or extend methods of classes. Valid compositions of feature modules (and according features) are defined in a feature model; we additionally presume the feature model to define an order for the valid composition of feature modules.

In Figure 1, we show a sample feature-oriented design using the Jak language [3]. The class refinement *Deque* of feature module *L2* in Figure 1b (class refinements are declared with the *refines* keyword, e.g., Line 7) adds a field *_depth* (Line 8) and a method *setElements()* (Lines 9-11) to the class *Deque* of feature module *L1* in Figure 1a. Class refinements add statements to methods by overriding these methods, e.g., method *insert front()* of the class refinement *Deque* (Fig. 1b) refines method *insert front()* of class *Deque* (Fig. 1a) by overriding. This method refinement calls the refined method using *Super* (Line 13) and adds statements. Feature module *L3* creates a new class *Container* that adapts the class *Deque* to be accessible under the name *Container*, and adapts method *Deque.insert front()* to be accessible under the name *add front* of the class *Container*. The composition result of *L1*, *L2*, and *L3* then includes both classes *Deque* and *Container*.

Refactoring	Transformation
Rename method	Changes the name of a method to reveal the method's purpose
Move method	Deletes the method and create a similar method in the class that the method uses the most
Add formal parameter	Adds a parameter to a method definition and according calls to provide additional information to the method body
Inline method	Replaces all calls to a method with the method's body and removes the previously called method

Table 1. Standard refactorings [16].

2.2 Refactoring

A refactoring describes a process of transforming the *system architecture* (members and classes) of programs but not their functionality [16]. A refactoring transforms members and classes to increase their reuse by removing conflicts with other classes. Refactorings facilitate subsequent extensions to the classes and improve understandability of members and classes [44, 33]. A single refactoring may transform a large number of members and classes and, thus, is very laborious to perform manually – for that, refactorings are supported by most integrated development environments,

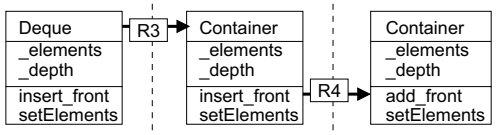

Figure 2. Refactoring methods and classes.

e.g., the Eclipse environment (v3.4.1) supports 23 different refactorings. In Table 1, we review some standard refactorings [16].

Refactorings expose different parameters where parameter values define the refactored members and classes [33], e.g., the parameters of a rename method refactoring are (1) the fully qualified name (*identifier*) of the method to rename and (2) the new method name.

In Figure 2a, we depict the class *Deque* composed from the feature modules of Figures 1a and b. Figure 2b shows the result of executing the refactoring *R3='rename class: Deque → Container'* which renames the class definition *Deque* and according references into *Container*. We show in Figure 2c the same class after executing the refactoring *R4='rename method: Container.insert front() → Container.add front()'* which renames the method definition *Container.insert front()* and every according call into *Container.add front()*.

3. Refactoring Feature Modules

We envision refactorings and sequences of refactorings as features in feature models such that a stakeholder can restructure the program she builds by selecting according refactoring features.[1] The refactoring is abstracted in the feature model with a name but must be defined in the features modules. We do this by adding refactoring definitions to feature modules.

In this section, we present *refactoring feature modules* a new approach that integrates refactorings into feature-oriented design; we also propose an extension to a feature-oriented language to implement our visions. We propose to define refactorings in refactoring units that become elements of feature modules – we call feature modules that encapsulate refactoring units *refactoring feature modules*. We describe which classes are transformed by a refactoring feature module and, finally, present *refactoring refinements* that gain synergy effects from combining refactorings and feature modules.

[1] We must ensure that feature modules with a refactoring are selected only when the refactoring can execute without errors but this is outside the scope of this paper.

```
1 class MyRenameClass implements
    RenameClassRefactoring {
2   String getOldClassId(){return "Deque";}
3   String getNewClassName(){return "Container";}
4 }
```

(a) refactoring feature module *R3*

```
5 class MyRenameMethod implements
    RenameMethodRefactoring {
6   String getOldMethodId(){return
      "Container.insert_front()->void";}
7   String getNewMethodName(){return
      "add_front";}
8 }
```

(b) refactoring feature module *R4*

Figure 3. Sample refactoring feature modules.

Definition of refactorings. A refactoring unit is a class that implements a refactoring interface. A refactoring interface corresponds to one refactoring, e.g., to the rename method refactoring, and describes the refactoring by declaring a getter method for every parameter of that refactoring. The getter method implementation in a refactoring unit then returns a value for that parameter. Together the parameter values of one refactoring unit fully identify the particular refactoring (instance).

As an example, we show in Figure 3 two refactoring feature modules *R3* and *R4* with a refactoring unit each. In the refactoring feature module *R3* of Figure 3a, a refactoring unit *MyRenameClass* implements the rename class refactoring, i.e., it implements the refactoring interface *RenameClassRefactoring* and defines the getter methods *getOldClassId()* and *getNewClassName()*. With these methods the refactoring unit provides the identifier of the class to rename and the new class name. The rename method refactoring of the refactoring feature module *R4* (Fig. 3b) implements the refactoring interface *RenameMethodRefactoring* and, thus, has to implement the methods *getOldMethodId()* and *getNewMethodName()*; the methods return values for the parameters of that refactoring.

In Figure 4, we show an equivalent feature-oriented design to that of Figure 1 but with the refactoring feature modules *R3* and *R4* (*R3* renames *Deque* into *Container*; *R4* renames *Container.insert front()* into *Container.add front()*). That is, the refactoring feature

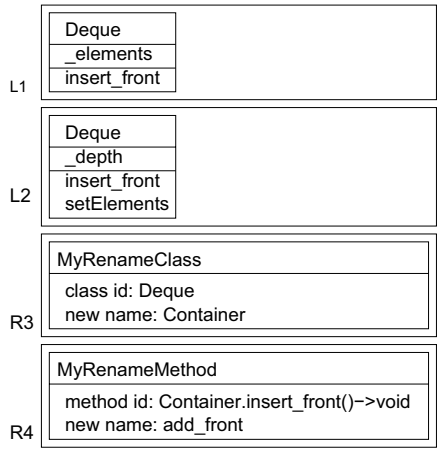

Figure 4. Refactoring feature modules.

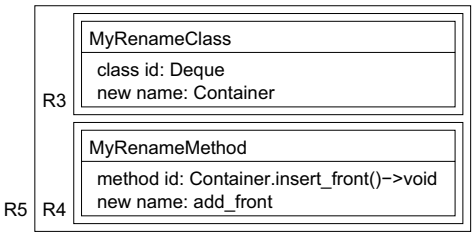

Figure 5. Composite refactoring feature module.

modules execute successively on the feature modules *L1* and *L2*. We use this figure as our running example.

Scaling refactoring feature modules. Refactoring units only execute without errors in a certain order. The feature model defines orderings for feature modules but not the order for units inside feature modules. Therefore, we allow only one refactoring unit in one refactoring feature module. However, this is not really a limitation: Feature modules scale and can be nested [3]. Thus, feature modules can encapsulate a number of refactoring feature modules – nested refactoring feature modules then can be ordered and build sequences of refactorings where a sequence of refactorings yields a (composite) refactoring [44].

In Figure 5, the refactoring feature modules *R3* and *R4* encapsulate a refactoring unit each. However, *R3* and *R4* are nested in a composite feature module *R5*. Henceforth, both refactoring feature modules *R3* and *R4* can be ordered in the feature model and the composite feature module *R5* represents the sequence of *R3* and *R4*.

Integrating refactorings into feature models. A refactoring feature module corresponds to a feature in the feature model. A feature model allows to select feature modules and refactoring feature modules but does not distinguish between them. That is, a user of this feature model can create a program by selecting features without knowing the program's implementation nor knowing the type of feature she selects (feature or refactoring).

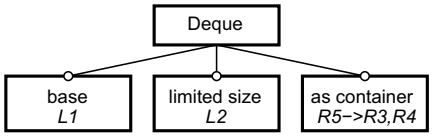

Figure 6. Feature diagram with refactoring features.

In Figure 6, we depict a sample feature diagram (graphical representation of a feature model) for our *Deque* data structure of Figure 4. In this diagram, features of the class *Deque* like *limited size* are related to refactorings like *as container* but, as mentioned, the feature model (and so the feature diagram) does not distinguish between features and refactorings. Furthermore, features are mapped to feature modules, e.g., *base* is mapped to *L1*, or to sequences of feature modules, e.g., *as container* is mapped to *R5* and, thus, to the sequence of *R3* and *R4*.

Control the set of refactored classes. We propose that refactoring feature modules should not transform members and classes which are created in feature modules the refactoring feature module precedes according to the feature model (so-called *bounded quantification* [31]). Refactoring feature modules should transform members and classes that are created in feature modules that the refactoring feature module follows according to the feature model. Bounded quantification of refactoring feature modules supports the multiple creation of classes and members, i.e., it avoids overriding for pieces of code that are created in different features but expose the same identifier. When different feature modules create code with the same identifier, a refactoring feature module that executes between both feature modules only refactors the one piece of code that the refactoring feature module follows according to the feature model. The code added by the subsequent feature module is not refactored. Bounding quantification of refactoring feature modules avoids effects on classes and members that are introduced after the refactoring feature module.

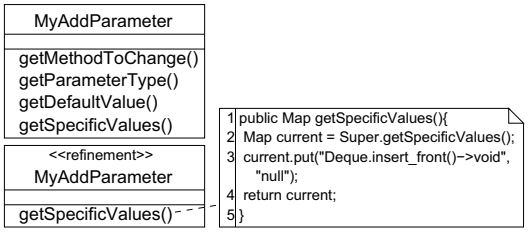

Figure 7. Refining refactoring units.

In the running example of Figure 4, the classes refactored by the refactoring feature modules *R3* and *R4* are limited to classes created by feature modules they follow. That is, *R3* refactors the code of feature modules *L1* and *L2* but not *R4* because *R4* executes after *R3*. An additional feature module *L5* (not depicted) introduces another class *Deque* like feature module *L1*. But, with bounding quantification, *R3* only renames class *Deque* of *L1*; *L5* then creates *Deque* without overriding *Deque* of *L1*.

Refining refactoring units. There are methods of refactoring units that should be extensible, e.g., the refactoring of adding a formal parameter to a method is, among others, parameterized with a default value for the formal parameter in according method calls and a collection of mappings from individual method calls to parameter values. Refining the latter method is beneficial because we can define additional mappings of method calls to parameter values and reuse the refactoring feature module.

To refine refactoring units, we apply the same rules that are proven beneficial for refining classes. That is, to refine a refactoring unit, we declare a refactoring refinement (declared with the *refines* keyword) of the same name in a subsequent feature module. The refactoring refinement then extends the refactoring unit and adds statements to the getter methods of the refined refactoring unit.

In Figure 7, a refactoring unit *MyAddParameter* adds a formal parameter into a method and parameter values into every caller of this method. One getter method *getSpecificValues()* of the refactoring unit returns a map of parameter values to method calls. The refactoring refinement *MyParameterRef* extends this getter method and adds further mappings of actual parameters to method calls. For that, the refinement overrides the method *getSpecificValues()*, calls the refined method using *Super* (Fig. 7, line 2), and adds mappings to the returned map (line 3).

Refactorings are composed with their refinements before the refactorings are executed and before feature modules are composed. That is, refining a refactoring unit delays the execution of the refactoring unit to the refinement. But, we argue that a feature should execute all of its transformations at the time defined in the feature model to avoid unintuitive side effects. A sample side effect could occur for code that was added by features after the refactoring unit but before the refactoring refinement – this code is refactored when the refactoring refinement executes (the refactoring gets delayed) but not when the refactoring unit is not refined. To avoid side effects of refactoring refinement, we restrict refactoring refinements to follow the refined refactoring unit directly. In the example of Figure 7, we do not allow any feature module to execute between the feature modules (not depicted) that encapsulate the refactoring feature module *MyAddParameter* and its refinement.

4. Application of Refactoring Feature Module

Scalable software libraries (SSL). Feature-oriented design reduces conflicts that occur when the system architecture (classes and members) of an object-oriented library precludes the library's reuse (so-called SSL [4]), e.g., when namespaces overlap by accident [35, 20, 22, 39]. For that, SSL strip the classes encapsulated in the library. But, if the stripped library classes still cause conflicts then the library still cannot be reused. Problems that arise when reusing libraries pertain incompatible member names, member locations, and subtype-relations of classes [20].

Refactoring feature modules restructure system architectures of libraries to avoid conflicts in applications that use the library. Thus, refactoring feature modules facilitate the library's reuse. However, refactoring feature modules do not reduce or change the functionality provided by the restructured library. Refactoring feature modules can be abstracted and used with a name but without knowing the implementation nor the refactoring effects; thus, even a stakeholder who does not know the SSL implementation can restructure the SSL (e.g., with a prepared sequence of refactorings that removes conflicts to one application). Valid compositions of refactoring feature modules are then, like valid compositions of feature modules, defined in the feature

model and executed as part of the feature-oriented configuration process.

The feature modules that separate class *Deque* in Figure 1 implement an SSL [4] and so does our equivalent running example in Figure 4. The refactoring feature modules *R3* and *R4* avoid adapter methods and classes (as in layer *L3* of Fig. 1) and, thus, simplify the composed library program. In particular, the composed library only encapsulates the desired class *Container* but no obsolete class *Deque* additionally.

Step-wise development (SWD). Software is developed in steps [54] and feature modules are used to portray these steps [3]. Feature modules then improve traceability of changes and understandability of the software [6]. However, class and method refinements rely on classes and members of preceding development steps (of feature modules they follow in the feature model) to apply extensions correctly. However, these classes or members maybe cumbersome to evolve [40, 19, 39, 50, 54]. Refactorings may restructure the members and classes in feature modules to facilitate the subsequently performed development step. But if the developer refactors existing feature modules to permit new extensions she fails the aim of *adding* evolutionary code (permanent edits to modules are considered problematic [15]). Refactoring existing feature modules to permit extensions may also add members and classes to even the first feature modules which then become more complex. Dig et al. observed that over 80 % of evolutionary changes are refactorings [12] so if feature modules portray development steps we must embed refactorings into SWD.

Refactoring feature modules restructure the program that is composed from feature modules without restructuring the sources of the feature modules. That is, refactoring feature modules avoid changes to existing feature modules but change the classes of the composed program. If development steps (and according feature modules) are iterated to re-engineer a program's system architecture the reviewer also benefits from feature modules that are simple for as many development steps as possible.

Feature modules in SWD are meant to extend but not to redefine the refined class/refactoring. When refactoring refinements replace any method of the refined refactoring unit, then the transformation of the refactoring unit is changed instead of extended, e.g., if a getter method is refined to return a different method that should be renamed then the transformation changes. But we also showed examples that benefit from refactoring refinement, like add parameter refactoring.

We advise to limit refactoring refinements in SWD. Refactoring refinements should extend only getter methods that define transformations for pieces of code that reference or depend on code that represents a refactoring parameter (refactored class or member). That is, a method that defines the renamed method should not be refined but a getter that maps parameter values can be refined.

Software product lines (SPL). SPL aim at covering a wide range of programs and can be implemented by composing modules in different combinations [41]; feature-oriented design is used to implement SPL [4, 30, 3, 52]. Features are mapped to members and classes that belong to certain functionality a stakeholder is interested in and that varies across programs. Features must provide extension points (classes and members) that can be refined in subsequently added feature modules. Extension points of one feature module exposed for single refinements then remain in the classes of every program where that feature module contributes. In the extreme case, members and classes covering the extension points for all refinements of the SPL (so-called *domain architecture* [18]) become the system architecture of every program with numbers of extension points unused. This approach is beneficial because a single domain architecture is easier to learn and extend than a number of system architectures [21]. But, this approach also is problematic: First, the system architectures (members and classes of a program) become overcomplex and hard to learn and reuse [42, 48, 44, 24]. Second, programs provide access to methods and classes that are extension points only and, thus, are not meant to be accessible outside the SPL.

Refactoring feature modules inline methods and classes that are extension points only, e.g., with the inline method refactoring (cf. Tab. 1). Thus, one domain architecture can be used for refinements but the system architecture of a single program is simplified automatically at composition time – inlining also increases the program's performance. The program with the simplified system architecture can then be analyzed and reused easier than programs with the domain architecture. Finally, methods and classes unmeant for access

can be inlined to become inaccessible; unused methods and classes can even be removed.

Experience showed that for an SPL one domain architecture which includes every extension point any refinement needs is easier to learn and extend than a number of system architectures with individual extension points [21]. But, programs derived from the domain architecture include numbers of unused extension points and, thus, are overcomplex.

We advise to use refactoring feature modules as final transformations in a feature model; that is, refactoring feature modules should adjust and streamline the program that is first composed from feature modules. This way, one domain architecture is easy to use and refine during development but every program's system architecture is simple as well.

Debugging. Refactoring feature modules complicate debugging because the classes of the debugged program differ from the classes inside the feature modules. That is, the members and classes of the debugged program correspond to members and classes in the feature modules that were restructured by refactorings. A change done in the program's classes, therefore, must be transformed back to the classes inside the feature modules. Hence, we need advanced debugging tools that keep track of the executed refactorings such that changes to the program's classes are triggered back to the feature modules automatically.

5. Implementation

We implemented refactoring feature modules as an extension to the Jak language which adds feature modules to Java [3]. In Figure 8, we show the components of our compiler and how these components interact. We reuse the *jampack* tool of the AHEAD tool suite[2] to compose Jak refinements. We implemented a plugin mechanism to support different refactorings for refactoring units where one refactoring corresponds to one plugin, e.g., the rename method refactoring is implemented in its own plugin. Our refactoring plugins follow a common interface and are loaded by the compiler when according refactoring units are executed. Inside every plugin, a refactoring interface (*RI*) is declared that corresponds to the refactoring, a refactoring interface which then is implemented by according refactoring units. The plugin further encapsulates the transformation of the refactoring. In Figure 8, we show the plugins for the refactorings rename class and move method which each declares a refactoring interface and the transformation to perform.

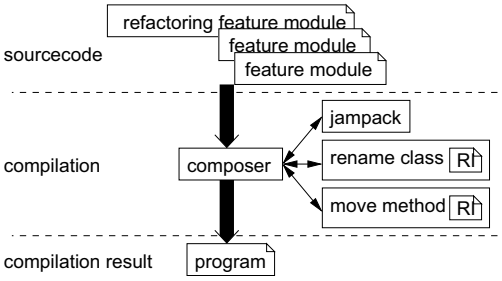

Figure 8. Compiler architecture.

The compiler successively composes the selected features modules and refactoring feature modules according to the order in the feature model. That is, the compiler executes refactoring feature modules alternately with class refinements of feature modules but not in advance. We do this because we found that for a number of refactorings we must compose refinements before we can execute the refactoring and for certain refinements we must execute refactorings first (refactoring and refinement are not distributive operations). For example, to inline a method we must compose the method first (refinements cannot be refactored into refinements of statements in Jak) and when a subsequently added feature module adds a new method of the same identifier then the method to inline should already be processed (we cannot push the execution of refactoring feature modules to the very end).

6. Related Work

Numerous researchers explore the role of transformations in feature-oriented design, materialized refactorings, and stepwise development. However, to the best of our knowledge no approach exists that combines both – feature-oriented design and refactorings. We report on the most related approaches now.

Languages. Different styles of *adapter modules* have been proposed [17, 27, 8]. Adapter modules allow to access an object using different interfaces (adapter and adapted interface) at the same time where the adapter modules only forward calls to according adapted methods. But, adapter modules may slow down the execution of the program, are tedious to write, and even

[2] http://www.cs.utexas.edu/users/schwartz/ATS.html

harder to understand and maintain [20, 32].[3] Adapter modules wrap single classes but do not extend feature modules as done by refactoring feature modules (Note that even adapter layers in LayOM [8] adapt single classes only.). Adapter modules, finally, imply maintenance for the application that uses the adapter when the adapted object (and so the adapter) changes [14]. *Comback layers* encapsulate adapter classes to undo refactorings on frameworks in an additional layer [10]. Refactoring feature modules do not create adapter classes but adjust the program classes.

Meta-programming may restructure a program or single classes [36, 22, 18, 23, 2]. The transformations existing meta-programming approaches provide exceed refactorings and may affect the functionality of programs – the refactorings, that we deal with, solely restructure programs but do not affect the program's functionality. Further, meta-programming approaches are not integrated into nor leverage from a feature-oriented design as refactoring feature modules do by refining refactorings.

JunGL [53] and *Generic Refactorings* [26] define refactorings in units. Others propose certain refactorings as *concepts of the programming language*, like adding parameters to methods [28]. As refactoring feature modules, all these approaches hide the developer from writing complex refactorings from scratch. But, the refactorings in these approaches transform modules permanently once executed, so these approaches do not deal with refactorings as configurable features. None of these approaches is integrated into a layered architecture nor leverages from that. Refactoring feature modules are integrated into feature-oriented design and can be selected and deselected. Refactoring feature modules can be refined to increase the refactoring feature module's reuse.

PARLANSE is a language to define program transformations [5, 6]. PARLANSE integrates edits according functionality and transformations. Although PARLANSE deals with edits as with transformations, PARLANSE is not integrated into a layered design and transformations gain no synergy effects like transformation refinements. PARLANSE transformations are defined by the developer and exceed refactorings because they may affect functionality. In PARLANSE, the definition of transformations is fragile because it is based on tokens [6], e.g., the transformation 'Δ*0:order→price@1:17*' changes token 17 in line 1 from *order* to *price*. Token-based approaches are additionally cumbersome for refactorings because referential integrity in the resulting program must be ensured manually. Refactoring units are pre-defined meta-proagrams that are parameterized only and automatically ensure referential integrity. Refactoring feature modules can be refined.

Collaboration-based phasing transforms models where transformations may be wrapped by other transformations [11]. Phases may also be optional which make them very similar to refactoring feature modules (although designed for model transformation we consider phasing to be applicable for programs). Phasing is not embedded in a layered design and transformations may affect the program's functionality. Refactoring feature modules do not affect functionality of a program. In contrast to phasing, refactoring feature modules do not introduce new programming concepts to refine refactoring definitions but reuse the refinement mechanism of feature-oriented design. Mens et al. formalize refactorings with *graph rewrite rules* and map graphs that represent programs to graphs that represent the refactored program [33]. They analyze the properties of refactorings on a formal level but do not investigate in refactorings as configurable items.

Transformation-based generators (TBG) are transformations that are successively executed on a program [7]. TBG can transform other transformations like modules and, thus, correspond to refinements; refactoring refinements use feature-oriented mechanisms to transform a refactoring. TBG are programmed by the developer from scratch and exceed refactorings, refactoring feature modules only parameterize a meta-program that implements a refactoring.

Multi-dimensional separation of concerns and *subject-oriented programming* divide a software into slices [19, 34, 39, 49]. Composition rules are meta-programs that integrate different slices. Composition rules rename and superimpose classes and methods of different slices but do not restructure the composed program. The developer of composition rules must ensure that the functionality of composed modules is not af-

[3] The adapter class *Container* in Feature *L3* of Figure 1 wraps class *Deque* and allows to access class *Deque* and method *Deque.insert front()* under the names *Container* and *Container.add front()* respectively. The adapter class *Container* needs additional maintenance and complicates the domain and system architecture.

fected; refactoring feature modules do not affect functionality by definition.

Aliases in *traits* adapt method names for method calls but do not change the system architecture of the composed program [45, 38]. Refactoring feature modules change the composed system architecture and can provide arbitrary refactorings in feature modules.

Feature-oriented refactoring [29, 51] and *aspect-oriented refactoring* [37] decompose a program into feature modules of a feature-oriented design and aspects respectively. In contrast, refactoring feature modules execute object-oriented refactorings on system architectures of programs which are composed from features.

Tools. *Compatiblity layers* is a tool-based approach to adapt libraries to applications that use the library where these applications rely on outdated interfaces of that library [14]. The integrated development environment of the library developer records edits and refactorings and replays the edits (without the refactorings) onto the outdated library version. Refactoring feature modules and compatibility layers allow to change a library to some extend without affecting the application that reuses the library. In compatibility layers, edits cannot be composed selectively which is possible for refinements in feature-oriented design. To replay a refactoring on different programs with compatibility layers one must compose every program first – this is impractical for a number of programs [25]. Refactoring feature modules integrate into feature-oriented design and, thus, edits can be selected and deselected. Further, refactoring feature modules refactor arbitrary programs at the time their according features are selected so refactoring feature modules can refactor programs that were never composed before.

The tool *MolhadoRef* records edits and refactorings on library programs and replays them in the application that uses the library [13]. Refactorings and edits cannot be selected by the user; refactoring feature modules and refinements in feature-oriented design can be selected and deselected.

Different researchers optimize the system architecture of programs with transformations: Smith proposes correctness-preserving transformations that are chosen by the user to improve performance, e.g., partial evaluation [47, 46]. Refactoring feature modules restructure programs to simplify reuse and integration of composed programs. However, by inlining classes and methods, refactoring feature modules may also improve the performance of the composed program. Smith's transformations, unlike refactoring feature modules, are not integrated into a layered design and cannot be refined. Critchlow et al. [9] optimize programs with refactorings toward metrics automatically. Refactoring feature modules are not selected automatically but manually in a feature model.

7. Conclusions

In this paper, we introduced refactoring feature modules. Refactoring feature modules are a new approach that combines feature-oriented design and refactorings. Refactoring feature modules adjust programs that are composed from features to encourage software reuse. Refactoring feature modules transform programs of feature-oriented design and leverage from the feature-oriented mechanism of refinements. In particular, refactoring feature modules facilitate reuse for classes that conflict to other classes and, thus, could not be reused before.

Acknowledgments

The authors thank Gordon S. Novak Jr. and Maider Azanza for helpful discussions and hints. Martin Kuhlemann is supported and partially funded by the *DAAD Doktorandenstipendium* (No. D/07/45661).

References

[1] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 265–279, 2005.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[4] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *SIGSOFT Software Engineering Notes*, 18(5):191–199, 1993.

[5] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 625–634, 2004.

[6] I. D. Baxter and C. W. Pidgeon. Software change through design maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, page 250, 1997.

[7] T. J. Biggerstaff. A new architecture for transformation-based generators. *IEEE Transactions on Software Engineering*, 30(12):1036–1054, 2004.

[8] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[9] M. Critchlow, K. Dodd, J. Chou, and A. Van Der Hoek. Refactoring product line architectures. In *Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[10] I. Şavga and M. Rudolf. Refactoring-based Support for Binary Compatibility in Evolving Frameworks. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 175–184, 2007.

[11] J. S. Cuadrado and J. G. Molina. A phasing mechanism for model transformation languages. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1020–1024, 2007.

[12] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[13] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[14] D. Dig, S. Negara, V. Mohindra, and R. Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 441–450, 2008.

[15] E. Ernst. Syntax Based Modularization: Invasive or Not? In *Workshop on Advanced Separation of Concerns*, 2000.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] J. A. Goguen. Parameterized programming and software architecture. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 2–10, 1996.

[19] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. *ACM SIGPLAN Notices*, 28(10):411–428, 1993.

[20] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 36–56, 1993.

[21] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[22] G. S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[23] G. S. Novak Jr. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering*, 23(7):401–417, 1997.

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

[25] C. W. Krueger. New methods in software product line practice. *Communications of the ACM*, 49(12):37–40, 2006.

[26] R. Lämmel. Towards generic refactoring. In *Workshop on Rule-Based Programming*, pages 15–28, 2002.

[27] K.-K. Lau, L. Ling, V. Ukis, and P. V. Elizondo. Composite connectors for composing software components. In *Software Composition (SC)*, pages 266–280, 2007.

[28] J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 108–118, 2000.

[29] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[30] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, 2001.

[31] R. E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2005.

[32] M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, page 203,

1997.

[33] T. Mens, N. V. Eetvelde, D. Janssens, and S. Demeyer. Formalizing refactorings with graph transformations: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[34] M. Mezini and K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 52–67, 2002.

[35] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136, 2004.

[36] M. Mezini, L. Seiter, and K. Lieberherr. *Component Integration with Pluggable Composite Adapters*. Kluwer, 2000.

[37] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122, 2005.

[38] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing duplication from java.io: A case study using traits. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 282–291, 2005.

[39] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *Workshop on Aspects and Dimensions of Concern*, 2000.

[40] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[41] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[42] W. Pree and H. Sikora. Design patterns for object-oriented software development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 663–664, 1997.

[43] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 1997.

[44] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[45] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, 2003.

[46] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[47] D. R. Smith. Kids: A knowledge-based software development system. In *Automating Software Design*, pages 483–514, 1991.

[48] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[49] P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[50] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Proceedings of the IEEE International Conference on Automated Software Engineering*, 8(1):89–120, 2001.

[51] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 191–200, 2006.

[52] S. Trujillo, D. Batory, and O. Diaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 44–53, 2007.

[53] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: a scripting language for refactoring. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 172–181, 2006.

[54] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.