Refactoring Feature Modules: Disciplined Generation of Reusable Modules

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.),

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von	Dipl.-Inform.	Martin Kuhlemann
geb. am	23.03.1982	in Magdeburg

Gutachter:
Prof. Dr. Gunter Saake
Prof. Don Batory, Ph.D.
Prof. Dr. Bernhard Rumpe

Ort und Datum des Promotionskolloquiums Magdeburg, den 14.11.2011

Kuhlemann, Martin:
Refactoring Feature Modules: Disciplined Generation of Reusable Modules
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2011.

Zusammenfassung

In der Softwaretechnik ist die Wiederverwendung von Code ein wichtiges Ziel, um Aufwand bei der Entwicklung neuer Programme zu sparen. Eine Möglichkeit den Code wiederzuverwenden ist, diesen in Module zu kapseln und diese Module dann zu verwenden. Das Entwickeln solcher wiederverwendbarer Module ist jedoch aufgrund des folgenden Skalierungsdilemmas schwierig. Einerseits soll ein Modul möglichst viel Funktionalität bereitstellen, um den Vorteil für denjenigen Programmierer zu erhöhen, der das Modul wiederverwendet. Andererseits sollen für ein Modul möglichst wenige unveränderliche Strukturentscheidungen getroffen werden, weil diese die Wahrscheinlichkeit erhöhen, in Konflikt mit den Entscheidungen zu stehen, die ein Programmierer bereits für sein zu entwickelndes Programm getroffen hat. Wissenschaftler fanden heraus, dass die beiden Ziele 'viel Funktionalität' und 'wenige unveränderliche Strukturentscheidungen' miteinander in Konflikt stehen, da ein Programmierer für ein Modul eine große Anzahl an unveränderlichen Strukturentscheidungen treffen muss, um anderen Programmierern Zugriff auf die Funktionalität des Moduls zu ermöglichen. Diesen Wissenschaftlern zu Folge kann ein Programmierer kein Modul implementieren, welches viel Funktionalität bereitstellt und gleichzeitig in einer großen Anzahl von Programmen ohne großen Aufwand wiederverwendet werden kann.

Ziel dieser Arbeit ist es, das Skalierungsdilemma von Modulen abzuschwächen. Dazu werden bereits bestehende Ansätze, die es schon jetzt erlauben, die Funktionalität von Modulen zu konfigurieren, so erweitert, dass im Folgenden die Struktur dieser Module konfiguriert werden kann. Technisch gesehen, werden Techniken der Software-Produktlinienentwicklung mit Refactorings kombiniert. Software-Produktlinien stellen einen Ansatz dar, um viele ähnliche Programme effizient zu entwickeln. Refactorings sind Programmtransformationen, welche die Struktur einer Software verändern aber nicht deren Funktionalität. Es wird von Fallstudien berichtet, die zeigen, dass Refactorings helfen, aber alleine nicht ausreichen, um Module in Programme zu integrieren, sondern dass Refactorings die bestehenden Techniken von Software-Produktlinien zu diesem Zweck ergänzen müssen. Weiterhin wird gezeigt, dass Refactorings zusammen mit Software-Produktlinientechniken helfen können, (a) Module bezüglich ihrer Funktionalität zu skalieren und zu konfigurieren sowie (b) Module bezüglich der bei ihrer Entwicklung getroffenen Entscheidungen hinsichtlich der Struktur zu konfigurieren. Das Ergebnis ist, dass die Funktionalität und die Struktur eines Moduls gleichzeitig an die Anforderungen eines Programmierers angepasst werden können. Für diese Anpassung wird jedoch kein Wissen über die Implementierung des Moduls benötigt, sondern nur Wissen über Domänenkonzepte. Zusätzlich wird sichergestellt, dass Entscheidungen bezüglich der Funktionalität eines Moduls nicht die Entscheidungen bezüglich der Struktur beeinflussen und umgekehrt.

Es wird demonstriert, dass Refactorings in Software-Produktlinien Code erzeugen und löschen können. Dabei zählen Beschreibungen von Refactorings nicht immer alle Code-Bestandteile auf, die sie verändern. Das hat zur Folge, dass durch die Analyse einer Beschreibung eines Refactorings nicht immer eindeutig bestimmt werden kann, welche Code-Bestandteile das Refactoring genau transformiert. Refactorings sind also nicht-monotone und nicht-aufzählende Programmtransformationen in Software-Produktlinien. Anknüpfend an diese Erkenntnisse werden im Rahmen dieser Arbeit verschiedene Techniken der Software-Produktlinienentwicklung analysiert und generalisiert, sodass diese Techniken auch nicht-monotone und nicht-aufzählende Programmtransformationen in Software-Produktlinien unterstützen. Genauer gesagt werden Techniken, die die Konsistenz zwischen bestimmten Modellen einer Software-Produktlinie und den Modulen dieser Software-Produktlinie prüfen, generalisiert. Überdies werden Techniken erweitert, welche es erlauben, die Fehler in der Funktionalität von Programmen einer Software-Produktlinie zu korrigieren. Innerhalb dieser Arbeit wird auch analysiert, wie Programme die Zeit reduzieren können, die (andere) Programme zur Ausführung einer Sequenz von Refactorings benötigen (z. B. um Programme einer Software-Produktlinie zu generieren). Abschließend wird das Refactoring von solchen Programmen untersucht, die Artefakte beinhalten, welche in mehreren Sprachen verfasst wurden.

Abstract

One goal in software engineering is to reuse code because reuse can reduce the effort of implementing new programs. One option to reuse code is to encapsulate code in modules and to use these modules henceforth. Modules are beneficial when they are reused in different programs; but, reusable modules are difficult to implement too because programmers face a module-scalability dilemma: On the one hand, a module shall provide a lot of functionality to increase the benefit for a programmer who reuses the module; on the other hand, modules shall expose few decisions regarding their structure to reduce the potential of conflicts with the structure that a reusing programmer expects. Researchers describe that both goals, to provide suitable functionality and structure, conflict because to provide access to a lot of functionality, a module must expose a lot of decisions regarding its structure. According to researchers who described the dilemma before, a programmer cannot implement a module that is full of functionality and is reusable in a high number of programs at the same time.

Our goal is to mitigate the module-scalability dilemma. We extend existing approaches, which already allow to configure the functionality of a module, in order to configure the structure of a module. Technically, we integrate techniques of software-product-line engineering with refactorings; *software product lines (SPLs)* allow to implement a number of similar programs efficiently; a refactoring is a program transformation that alters the structure of programs but not their functionality. In case studies, we observed that refactorings help but alone do not suffice to integrate modules with programs, but that refactorings complement the techniques that are already in use in SPLs. We found out that refactorings together with SPL techniques can help (a) to configure modules with respect to functionality and (b) to configure modules with respect to their structure. As a result, users can configure the functionality as well as the structure of a module to be suitable for a reusing program.

We demonstrate that refactorings in SPLs (a) create and remove code and (b) do not always enumerate all the pieces of code they create and remove (i.e., by analyzing a refactoring description, one does not always know all the pieces of code which the refactoring transforms). Thus, refactorings are nonmonotonic and nonenumerative program transformations in SPLs. With this insight, we analyze and generalize different techniques of SPL engineering such that they support nonmonotonic, nonenumerative program transformations of refactorings in SPLs: First, we generalize techniques to verify consistency between certain models of SPLs and the modules of respective SPLs when these SPLs involve refactorings. Second, we generalize techniques that support programmers to correct errors in the functionality of programs of SPLs when these SPLs involve refactorings. Third, we analyze how a tool can reduce the time that a (second) tool needs to execute a sequence of refactorings (e.g., to generate a program). Finally, we use refactorings to configure programs of SPLs that include artifacts written in multiple languages.

Acknowledgments

I would certainly not have finished this thesis without a number of persons whom I thus owe a lot. First, I thank my wife, my family, and my friends. It was very important for me to know that I can always count on them. Thank you folks.

Second, I thank Gunter Saake for his constant and holistic support of my research. Gunter provided me with a great research atmosphere in his group where I was free in choosing the research challenge I am interested in the most. Gunter also gave me the opportunity to collaborate closely with other researchers by visiting them; even if this meant to swap lectures and being absent for a semester. Gunter, I enjoyed being part of your group; thank you for this opportunity!

Third, I thank Don Batory whom I visited at the University of Texas at Austin. Don challenged me to be patient with research results and to spend more time on thinking on results of my own research; that is, Don encouraged me to find and tackle the "underlying reasons" of problems. Don further encouraged me to implement the most scalable and elegant solution I could imagine; even if I already had implemented a (less strong and less elegant) prototype. Especially, the frequent discussions with Don helped me to improve my ideas. Don, I enjoyed our discussions a lot; thank you!

Fourth, I thank Ralf Lämmel, Krzysztof Czarnecki, and my colleagues at the Universities of Magdeburg, Austin, and Waterloo; they were always available for intensive, long-term research discussions. Among them, I particularly want to thank Christian Kästner, Sven Apel, Norbert Siegmund, and Marko Rosenmüller for the great discussions we had. I thank Andreas Lübcke and Christian Kästner for supporting my research stays by swapping lectures.

Finally, I thank my students.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
FOP	Feature-Oriented Programming
MLR	Multi-Language Refactoring
NFP	Nonfunctional Property
GPL	Graph Product Line
OOP	Object-Oriented Programming
RFM	Refactoring Feature Module
SPL	Software Product Line
SQL	Structured Query Language

1. Introduction

Programmers reuse code to reduce the effort of implementing new programs [Par76]. Programmers who decompose code into distinct modules are advantaged [Par78] because they might be able to implement a piece of functionality in a new program by copying a single entity – the module – from an existing program. Programmers who do not decompose code must extract according code from an existing program first or must reimplement the functionality without reuse.

A module is difficult to develop because programmers face a fundamental dilemma [Big98, CHSV97, CL01, Her08, HM07, Mey97, OT00, TOHS99, Weg90]: On the one hand, programmers can implement a module which is large-scale and valuable (i.e., full of functionality) – but, then this module cannot be reused in many programs because the module is difficult to integrate with these programs. On the other hand, programmers can implement a module which is small and easy to integrate with reusing programs – but, then this module is less valuable and reusing it does not pay off. Integration problems occur among other things because programmers must decide on how to structure code to implement functionality, and this structure can conflict with the code structure of a reusing program (e.g., the allocations or names of pieces of code) [Par72]. Researchers found that, in general, programmers cannot implement a module that is full of functionality and easy to integrate with a high number of programs at the same time.

Research on domain engineering and *software product lines (SPLs)* tackled the problem of missing module functionality [BSST93, vdS04]. Domain engineering studies the differences and commonalities between programs of a domain. During domain engineering, engineers discover distinguishable characteristics, called *features*, of programs of a domain [CE00]; these engineers also relate features to each other and describe them as common among programs or as variant. An SPL is a set of programs of a domain that can be developed from a shared code base [CE00]; the features that contribute to multiple programs enable code reuse [CN06]. Domain engineering and SPLs allow a user to configure a module to provide the functionality he/she desires.

In this thesis, we shed light on how programmers can configure the structure of a module to aid reuse (e.g., to configure the names of modules). Specifically, we extend existing techniques of SPL engineering to generate configured modules. As a new opportunity, programmers can now configure a module with respect to its structure to be reusable in programs it could not be reused in as is before. As one follow-up challenge, we had to cope with higher complexity of verifying consistency between (a) combinations of features which a model describes to be legal for the programs of an SPL and (b) supported combinations of modules that implement these features. We observed that verifying this consistency manually became hard even for small case studies; thus, one could argue that with our techniques we push the boundaries of SPL technology to a point, at which powerful mechanisms of program generation can only be managed safely with concepts and tools as we present in this thesis. A user in our context might be a programmer of a program which shall reuse a module of the module SPL, or might be a person who just knows that a program requires a module in a certain structure.

1.1. Who Should Read this Thesis?

Researchers interested in modular programming should read this thesis. When programmers are asked by different customers to develop a set of modules in the same domain that differ in their functionality and their structure, then these programmers basically either (a) can negotiate with the customers to accept a common structure, (b) can refuse business with certain customers, and/or (c) can develop the modules independently from scratch for each customer. Note that if customers want to integrate the module with legacy programs, they rarely accept a structure that differs from what they requested because they are often times afraid of changing the legacy programs. We introduce techniques which allow such programmers to efficiently develop a set of modules that differ with respect to functionality and structure. Note finally that none of the above options is easy to implement.

1.2. Contribution

In this thesis, we make the following main contributions: We extend techniques of SPL engineering such that users can configure the structure of modules, we generalize existing concepts of safe composition for SPLs, and we evaluate practical issues of configuring the structure of modules.

1. We extend techniques of SPL engineering such that users can configure the functionality *and* the structure of a module. Thereby, users will be able to define configurations without knowing anything about the module's implementation, because we provide techniques that allow users to configure the structure of a module based on the concept of features; techniques that allow users to configure functionality based on features already exist. Our approach ensures that decisions regarding the functionality of a module do not affect decisions regarding the structure of this module, and vice versa.

   Specifically, the techniques we propose allow users to configure programs and modules using selectable refactorings. A refactoring is a program transformation which alters the structure of programs but not their functionality [Opd92].[1]

[1]The term *refactoring* is actually defined with respect to the *behavior* of a program. In the remaining thesis, we use the term *functionality* as a synonym for *behavior* because (a) the term *functionality* suffices to understand our discussions, (b) the term *functionality* helps us to describe basic relations between refactoring and SPL mechanisms, and (c) literature on SPLs uses both terms synonymously, too.

We call our technique *refactoring feature modules (RFMs).* We demonstrate that RFMs are feasible to remove incompatibilities between modules and programs that shall reuse the modules, and to configure nonfunctional properties of programs such as performance.

2. We generalize existing concepts of safe composition, which verify that the implementation of an SPL is consistent with its variability model. Existing concepts verify consistency of SPLs built from program transformations that (a) only add code or only remove code, and that (b) enumerate all the pieces of code they add or remove (i.e., by analyzing a program transformation, one knows *all* the pieces of code it transforms). We generalize these concepts to verify consistency of SPLs built from program transformations that create *and* remove code, and that do *not* always enumerate all the pieces of code they transform.

3. We present techniques which support programmers in practical issues of SPLs when these SPLs involve RFMs. In particular, we present techniques for how to correct errors in functionality of programs of SPLs; programs which have been generated using RFMs. We present techniques for how to reduce the time a tool needs to generate programs of SPLs. We discuss requirements and challenges to apply refactoring techniques to those programs, of which each is written in more than one language.

1.3. Disclaimer

In this thesis, we present our concepts for programs written with concepts of *object-oriented programming (OOP)* and for SPLs written with concepts of *feature-oriented programming (FOP).* Throughout this thesis, we exemplify our concepts using the OOP language Java [GJSB05] and using the FOP language Jak [BSR04]. Please note that the presented concepts apply for *arbitrary* languages and programming paradigms for which we can describe refactorings in modules; we exemplify our concepts for Java and Jak only for understandability.[2] Furthermore, for understandability reasons, we present our concepts using a simple running example of a *graph product line (GPL)* (developed and proposed as a benchmark study for SPL approaches elsewhere [LHB01]) and with simple standard refactorings (described elsewhere [Fow99]); please note, however, that our approach can scale to arbitrarily complex SPLs and use arbitrarily complex refactorings.

[2]We prototypically implemented parts of the concepts presented in this thesis. For our implementation, we reused several modules and extended existing tools such as Sat4J (`http://www.sat4j.org/`; accessed: July 16,2011), JastAdd [EH07], or tools from the AHEAD tool suite (`http://www.cs.utexas.edu/users/schwartz/ATS.html`; accessed: July 16,2011). We distinguish our work from these foundations and other related work in Sec. 3.3 and Chap. 7. In the remaining sections, we focus on *concepts* and thus omit discussions on our implementation unless necessary.

1.4. Outline

This thesis is structured as follows[3]: In Chapter 2, we introduce the basic concepts we rely on in this thesis. In Chapter 3, we lead to the main problem we tackle in this thesis. In Chapter 4, we introduce the concept of RFMs and report on case studies we conducted with RFMs. In Chapter 5, we discuss inconsistencies between the variability model of an SPL and its implementation, and we discuss how to detect these inconsistencies. In Chapter 6, we present concepts that help to use RFMs in SPLs. In Chapter 7, we discuss our concepts with respect to related research. In Chapter 8, we summarize the concepts and insights of this thesis and give possible lines of future work.

[3] Chapter 3 shares material with [KBA09, KSA10]. Chapter 4 shares material with [KBA08, KBA09, KKAS11, SKAP10]. Chapter 5 shares material with [KBK09]. Chapter 6 shares material with [KLS10a, KS10b, SK10, SKSL11]. Chapter 7 shares material with all papers referenced in this footnote.

2. Background

The topics in this chapter cover the *basic* concepts we rely on later in this thesis; specialized background is given in later chapters, as needed. In this chapter, we review basic concepts of modular programming (cf. Sec. 2.1) and SPLs (cf. Sec. 2.2). Later in this thesis, we extend the concepts of SPL engineering that are implemented with modular programming. Accordingly, we now review the implementation techniques OOP (cf. Sec. 2.1) and FOP (cf. Sec. 2.2.2). Finally, we review refactoring (cf. Sec. 2.3) because as part of our approach, we integrate refactorings with SPL-engineering techniques.

2.1. Modular Programming

Researchers argue that code of a program should be decomposed into modules to reduce this code's complexity, to implement different parts of a program in parallel, and to increase the reuse of code [Dij69, Dij82, Par72, SMC74]. A module has some name, some implementation of the module functionality, and can have a description of the module functionality.

The implementation of a module's functionality comprises commonly a set of algorithms that compute results for given inputs. The implementation can be written in languages of different programming paradigms such as languages of logical programming [CR96], functional programming [Jon03], or imperative programming [GJSB05]. In this thesis, we concentrate on modules implemented with the imperative programming language Java (cf. Sec. 2.1, p. 6).

The description of a module's capabilities commonly is called an *interface* and is used to abstract from the module implementation [GHJV95, Mey97, PBvdL05]. In this thesis, we need to distinguish user interfaces from *application programming interfaces (APIs)*:

- User interfaces describe modules with respect to presentation elements that the modules provide. Users can use these elements to interact with the modules. Sample presentation elements are dialog boxes, menus, or terminals [Mye88].
- An API describes a module with respect to algorithms, which the module encapsulates and which can be referenced from code outside the module (we call such outside code an *environment*) [BCK06, CE00, Gro04]. APIs include method signatures which define how environments can call a method in this module (i.e., with which input values) and what they can expect from a method as a result [BCK06]. Further, APIs describe modules with respect to states

```
1 package gpl;
2
3 public class Vertex {
4   public String name;
5   public void display(){
6     ...
7     System.out.println();
8 } }
```

Figure 2.1.: *Excerpt of a class from the GPL (package declaration inserted for explanatory reasons) [adapted from LHB01].*

which these modules maintain. Sometimes even functionality descriptions of algorithms are part of an API [BCK06, CE00].

APIs of a module allow a module programmer to hide implementation details of a module to some extend from environments; for example, when APIs do not expose every implementation detail of a module. As a result, the programmer can change those details of a module without the need to update environments on behalf of this change [BCK06, Mey97, Par72, Sny86, Str91]. APIs also allow those programmers to replace a module by a different one when both provide the same API [GHJV95, PBvdL05, Sny86].

APIs can be described separately from the module's code in a named artifact (we call such named artifact *API artifact*). API artifacts can be defined with concepts of the programming language the module is implemented in (e.g., the interface concept in Java and the virtual concept in C++ [GJSB05, Str91]), or API artifacts can be defined with dedicated API description languages [DK76, Gro04, vdS04]. In the remaining thesis, we say that a module *implements* an API artifact when this module provides an implementation for the algorithms and states described in the API artifact.

OOP in Java

In this subsection, we review concepts of the OOP language Java at a low and technical level (e.g., we review inheritance and overriding); we do so to avoid confusion when we describe and detect possible errors of refactorings later. In case, the reader is familiar with OOP concepts and Java, we advise to continue with the Section *Object-Oriented Design Patterns* on page 9.

OOP is one approach to decompose programs into modules. Large-scale modules of current mainstream OOP languages are called *packages*; packages encapsulate other packages or small-scale modules [GJSB05, Int07, Mey97]. Small-scale modules of current mainstream OOP languages are called *classes*; classes encapsulate class members [Mey97]. Class members can be variables, methods, and nested classes [GJSB05]. In Figure 2.1, we show parts of a package gpl; this package contains a class Vertex; Vertex encapsulates the member variable name (Line 4) and the method display (Lines 5-8). In Java, API artifacts are defined as special classes that have no implementation for the methods they declare.

Objects are runtime entities instantiated from classes [Mey97]; that is, objects provide a field for each member variable of the class they are instances of. The type of an object associates this object to the class it is an instance of and the methods of this class respectively [Mey97]. Objects are instantiated by constructor methods (constructors for short). In Figure 2.1, all objects instantiated from class Vertex are of the same type (i.e., they all have a field name and a method display because Vertex defines according members).

An object can access methods and fields of other objects by referencing these objects with variables, or an object can access methods and fields of itself by referencing itself using the pseudo variable this [GJSB05, Str91]. Methods and fields that correspond to class members modified with static (static class members) can be accessed additionally by referencing the class, which hosts the members, instead of an object [GJSB05]. In Figure 2.1, Line 7, the method display among others references the field out of class System without a System object (out is static); this field references an object which then is used to call the method println.

Modules and their code respectively reference each other unambiguously using names; for example, a class has a name which other pieces of code use to reference this class. Modules further introduce *scopes* for the names they encapsulate such that names of pieces of code inside a module need not be unambiguous in the whole program but only within the scope of that module [Gro04, Str91]; for example in Figure 2.1, the scoped name of the class named Vertex, which is nested inside a package gpl, is gpl.Vertex and there might be more classes Vertex in other packages than gpl [Gro04, Str91]. As a result, scoped names are unambiguous in a program.

Only methods are an exception to the unambiguous-name constraint of scoped names; that is, different methods within the same scope might have equal names and thus equal scoped names [Mey97]. For that, in order to reference a method unambiguously, the scoped name of that method must be combined with parameter objects this method accepts – the types of these objects determine the method to execute; the combination of a method's scoped name, all types that the method's parameter variables *have* (i.e., all types of objects that the variables declare to reference), and the method's return type is called the *signature* of that method [GHJV95, GJSB05, Gro04]; signatures identify methods unambiguously. In Figure 2.1, the signature of method display is gpl.Vertex.display()::void. For homogeneity of our descriptions, we add the types that member variables have to the scoped names of these variables, too.

OOP in Java introduces the class-based reuse mechanism of inheritance and the object-based reuse mechanism of forwarding [Weg90]. Inheritance allows a class called *subclass* to reuse (inherit) the members of a class called *superclass* [Mey97, Weg90]. Objects of a class thus provide methods and fields as defined in that very class or any of its superclasses. For that, subclass objects can be assigned to variables that *have the type of* the subclass (i.e., that declare to reference objects of the subclass) *or* any of its superclasses [Mey97]. If an object of a subclass

```
 1 public interface GraphElement {
 2   public void display();
 3 }
```

```
 4 public class SpVertex extends gpl.Vertex implements GraphElement {
 5   private String predecessor;
 6   private int dweight;
 7   public void display() {
 8       System.out.print( "Pred " + predecessor + " DWeight " + dweight + " " );
 9       super.display();
10    } }
```

Figure 2.2.: *Subclasses in OOP.*

is assigned to a variable which has the type of a superclass of the subclass, a method of the subclass is executed instead of a method of the superclass that was called using the variable *if* both methods have equal signatures (such subclass method is said to *override* the superclass method [GR83, GJSB05, Int07, Mey97]). In Figure 2.2, SpVertex declares to inherit class members from gpl.Vertex (keyword extends, Line 4) and so SpVertex objects provide a field name. SpVertex further declares to implement the API artifact GraphElement. At runtime, SpVertex objects can be assigned to variables that have either the type SpVertex or the types gpl.Vertex or GraphElement (because gpl.Vertex and GraphElement are types of superclasses or API artifacts of SpVertex). However, in any case, a call with such variables (which reference SpVertex objects) to a method display will execute SpVertex.display()::void. In the example of Figure 2.2, the SpVertex method does not replace the gpl.Vertex method when method display is called with SpVertex objects; but, SpVertex.display()::void calls gpl.Vertex.display()::void (keyword super, Line 9) and thus extends this method.

Forwarding allows a method of an object to provide functionality by calling a second method (possibly of a referenced object) [Weg90]. Different methods of possibly different objects may forward to the same method, and methods of one object may forward to different methods of different objects. Commonly, forwarding objects reference the objects, which they forward to, using a field (which corresponds to a member variable in the class).

Members of a class can be modified with the access modifiers public, protected, and private [GJSB05]. A member modified with public can be referenced from every piece of code in the class itself and in the environment of this class (i.e., in code outside this class). A member modified with protected can be referenced only from code (a) of the class hosting the member and (b) of every subclass of the class hosting the member. A member modified with private can be referenced only from code of the class hosting the member [GJSB05, GR83]. Unmodified class members can be referenced only from code of the same package or from code of classes that inherit from the member's host class [GJSB05]. In Figure 2.2, method SpVertex.display()::void is modified with public and thus can be referenced from every piece of code in the program; SpVertex.dweight::int is modified with private and thus cannot be referenced

```
public interface VertexI{
  public void show();
}

//wrapper
public class ADT implements VertexI{
 private Vertex wrappee;
 public void show(){
  wrappee.display();
 } }
```

```
//wrappee
public class Vertex {
 public String name;
 public void display() {
  ...
  System.out.println();
 } }
```

Figure 2.3.: *Wrapper design pattern implemented for GPL classes.*

from code outside class SpVertex.

To reuse OOP code at a large scale, programmers may implement frameworks or libraries. An OOP framework is a program of which users may alter functionality by extending dedicated classes [CHSV97, JF88]; different sets of user classes then generate different programs which all reuse the framework. An OOP library encapsulates classes and packages to be reused in programs [Weg90].

Object-Oriented Design Patterns

We rely on a common understanding of object-oriented design patterns several times in this thesis and especially on a common understanding of the pattern Wrapper, which we exemplify below.

There are different ways, how programmers can decompose their object-oriented code into modules [TOHS99, Tor04]. Object-oriented design patterns are descriptions of best-practice decompositions of object-oriented code with respect to a recurring development task [GHJV95]. Researchers cataloged object-oriented design patterns such that an unexperienced programmer is capable to decompose his/her code *best* into modules from beginning. Researchers identified object-oriented design patterns at different levels of abstraction from code [Chr04, GHJV95, Woo97, Zim95]; the object-oriented design patterns we deal with in this thesis are at the level of classes; that is, the patterns describe what functionality a class should encapsulate and how classes should be connected (inheritance/forwarding) to solve a recurring development task.

A *wrapper* is a well-known object-oriented design pattern [BCK06, GHJV95]. Wrapper describes a proven way to provide an additional API for an object; thereby, an object of a class called *wrapper class*, which has a desired API, forwards calls to a referenced object of a class called *wrappee class*, which has an undesired API.[1] At runtime, wrapper objects then replace wrappee objects and accept calls with their (desired) API. As (a) wrapper classes can be structured differently than their respective wrappee classes, (b) wrapper objects can replace wrappee objects at run-

[1]There are different implementations of Wrapper [GHJV95] but we limit ourselves at this point of the thesis to one for understandability.

time, and (c) wrapper objects can provide their wrappees' functionality (through forwarding), wrappers simulate a different API for wrappees.

In Figure 2.3, we extend the code of our GPL running example to make ADT objects wrap Vertex objects (please ignore VertexI for now). We defined a member variable wrappee in class ADT in order to assign wrappee objects to it (Line 7). We defined a method show in ADT (Lines 8-10) that forwards its calls to method display of the referenced Vertex object. As a result, ADT objects provide a different API for respective Vertex objects because ADT objects allow programmers to reference Vertex objects using variables, which have the type ADT, and allow programmers to access the display method using name show.

2.2. Domain Engineering and Software Product Lines

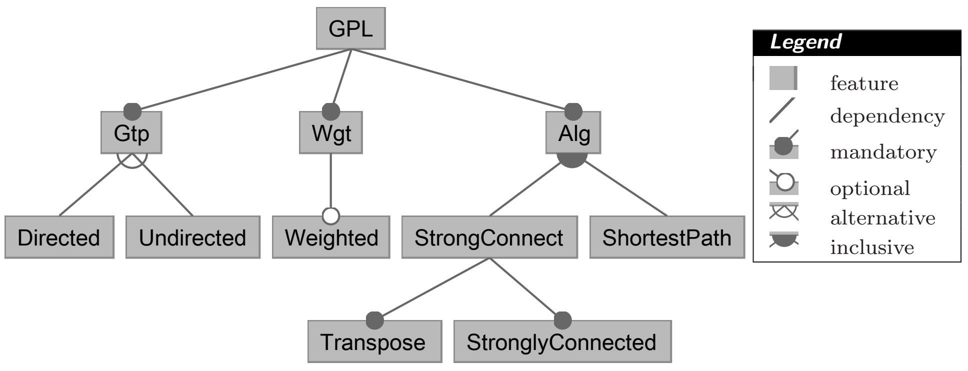
An SPL is a set of programs of a domain; those programs in turn are called *products* of their SPL. Products of an SPL cover the same market segment, share features, and differ in features [CE00, CN06]; a *feature* in an SPL product is a distinguishable program characteristic important to some user [CE00]. In contrast to an SPL, a program family comprises different programs that share code [Dij69, Par76]. SPLs can (but do not have to) be implemented as program families such that different SPL products share code, too [BSR04, CE00, CN06]. In this thesis, we extend the work of others and so in line with them we use the term *SPL* to refer to a set of programs of a domain that share code.

The development of an SPL is commonly called *domain engineering* or *SPL engineering* [CE00]. To develop an SPL, domain engineers first analyze the domain of the SPL and thereby model the variability of programs of that domain with respect to features [CE00]. After that, a programmer can implement different SPL products in parallel by implementing features which these SPL products share. Overall, the effort to implement SPL products in parallel is intended to be less than the effort to implement the same SPL products independently from each other [CE00, Par76]. We review domain analysis in Section 2.2.1 and domain implementation in Section 2.2.2.

Though SPLs provide benefits, not every program should be implemented with SPL-engineering techniques from beginning [CK02]. In general, the time programmers need to implement a stand-alone program is less than the time they need to implement the same program in parallel with other programs of an SPL. SPLs first pay off, when a number of similar programs are demanded. In this thesis, we are interested in the development of sets of similar programs.

2.2.1. Domain Analysis

As the first step to implement SPLs, domain engineers commonly perform a domain analysis. One approach to analyze a domain is to perform a *feature-oriented domain analysis* [KCH^{+}90]. Engineers, who perform a feature-oriented domain analysis, analyze a domain in three steps: context analysis, domain modeling, and architecture modeling. During the context analysis, engineers define the scope of the analyzed



(*ShortestPath* implies *Directed* and *Weighted*)

Figure 2.4.: *Feature diagram of the GPL [adapted from LHB01].*

domain by relating it to other domains [KCH+90] (i.e., they define the boundaries of the analyzed domain); the engineers further define rough APIs of future SPL products [KCH+90]. During domain modeling, engineers identify the features of future SPL products and relate the features to each other [KCH+90]. During architecture modeling, engineers define the structure of future SPL products [KCH+90].[2]

One major outcome of the feature-oriented domain analysis is a feature model. A feature model relates the features of (future) SPL products to each other, groups the features, and defines meaningful combinations of the features (i.e., not all combinations are meaningful [BO92]) [CE00, KCH+90]. Features can be declared to be mandatory, optional, or alternative for programs of that domain [KCH+90]. A feature model finally can be used by SPL users to select an SPL product by selecting its features [CE00, KCH+90].

A feature diagram is a graphical notation of a feature model [CE00, KCH+90]. In Figure 2.4, we depict a feature diagram for a simplified version of a feature model of our GPL running example [adapted from LHB01]. The diagram shows a tree in which features are represented by rectangles and feature relations are represented by straight lines, circles, and semicircles. The root feature generally indicates the modeled domain; for example, in Figure 2.4, the feature diagram models the domain of graph data structures (i.e., the domain of the GPL). Straight lines denote grouping relations such that all child features of a parent feature are members of one group. Straight lines further denote dependencies of child features toward their parent feature in the tree; for example, in Figure 2.4, features *ShortestPath* and *StrongConnect* are defined to be child features of feature *Algorithm* (*Alg*) and thus are grouped; at the same time, *ShortestPath* and *StrongConnect* can only be selected when *Alg* is selected, too. Circles denote dependencies of parent features toward an individual child feature. Filled circles denote that a parent feature depends on its child feature;

[2]Sometimes the architecture-modeling phase is described as its own step after domain analysis and domain implementation [CE00] – the differences between both options are not important for this thesis.

for example, in Figure 2.4, *StrongConnect* can only be selected when its child feature *Transpose* is selected as well. Empty circles denote that a parent feature does not depend on its child feature; for example, in Figure 2.4, the feature *Weight* (*Wgt*) can be selected independently of its child feature *Weighted*. Semicircles denote dependencies of parent features toward a group of child features. Filled semicircles denote that a parent feature depends on that *at least one* in the group of its child features is selected; for example, in Figure 2.4, *Alg* depends on that *StrongConnect* is selected, or *ShortestPath*, or both. Empty semicircles denote that a parent feature depends on that *exactly* one in the group of its child features is selected; for example, in Figure 2.4, feature *Gtp* depends on that feature *Directed* is selected, or *Undirected*, but not both.

A feature diagram cannot depict all dependencies between features described in a feature model because feature diagrams must expose a tree structure but dependencies between features in feature models need not [KCH+90]. Relations which cannot be described in feature diagrams must be described separately [CE99a, KCH+90]. For the diagram of Figure 2.4, we describe separately (underneath the diagram) that *ShortestPath* can only be selected together with the features *Directed* and *Weighted*.

2.2.2. Domain Implementation

Programs of an analyzed domain can be finally implemented as products of an SPL in a domain-implementation phase [CE00]. SPLs can be implemented with annotative or compositional approaches [KAK08, KRT97], or a combination of both [KAK09]. In annotative approaches, programmers commonly implement a single program first with all features described in the feature model. After that, the programmers annotate each piece of code with respect to the features it implements. When a user, finally, selects desired features of an SPL product, code of nonselected features is removed and code of selected features remains. Annotative SPL approaches include #ifdef preprocessors [SC92, Str91] and user-interface approaches [BOT07, KAK08].

In compositional approaches, programmers implement a module or a set of modules for every feature in the feature model [KAK08]. These feature-related modules are synthesized once a user selects the features he/she desires for his/her SPL product. The result of synthesizing feature-related modules then is an SPL product with the code of the desired features. To implement the feature-related modules with less effort, programmers commonly target at a common structure for all SPL products and thus for all feature-related modules [Bat07b, BOT07, CAK06, CN06, JF88, PBvdL05]. Compositional approaches used for SPLs include frameworks [BMMB00, CHSV97, JF88], aspect-oriented programming [Gri00, HC02], and FOP [BSR04].

In this thesis, we extend compositional SPL approaches by modules that describe refactorings. In particular, we exemplify our concepts on top of FOP.

Feature-oriented programming in Jak

Feature-oriented programming (FOP) is one approach to decompose programs into modules, which are dedicated to features; these modules are then called *feature modules* [BSR04, Pre97]. For the feature-driven decomposition of programs, it does not matter whether these programs include artifacts written in programming languages, natural languages (e.g., documentation), grammar languages, XML languages [DLT00, KKKS08a, KWDE98, Lin95, Mar05, SKL06], or other languages; FOP techniques apply to all of these artifacts uniformly (named *principle of uniformity* of FOP [BSR04]) [ADT07, AKL09, BSR04, SKS$^+$08, TBD06].

Jak is a language extension for Java that supports feature modules [BSR04]. In Jak, a feature module is a program transformation that is dedicated to a single feature; this feature module then adds (copies) code in a way to its input program that the generated program implements the feature. A feature module encapsulates classes and class refinements.[3] A class inside a feature module is added to the input program of this feature module. A *class refinement* inside a feature module is a program transformation on its own; it is executed on a single class of the feature module's input program. Class refinements encapsulate class members and method refinements. A class member inside a class refinement is added to the class of the input program which the class refinement transforms. A *method refinement* inside a class refinement is a program transformation on its own; it is executed on a single method of the class of the input program which the class refinement modifies. Method refinements encapsulate statements that are added to the refined method.

Feature modules execute consecutively to generate a program; executing them in different combinations, however, yields different programs – the products of an SPL. The order in which feature modules can be executed is oftentimes fixed and predefined for a complete SPL [AKGL10, Bat05, Bat06, BO92, BSR04, KKB08]. In Figure 2.5a, we show the feature modules *Directed* and *ShortestPath* of our GPL running example. *Directed* is a program transformation that transforms its input programs to make them provide the feature of directions in graphs. *ShortestPath* is a program transformation that transforms its input programs to make them provide the feature of shortest-path algorithms for graph programs. The modules are executed in top-down order such that the input program of *Directed* is empty and the input program of *ShortestPath* is the generated program of *Directed*.

In Figure 2.5a, *Directed* encapsulates a class Vertex and adds a copy of this class to the empty *Directed* input program. *ShortestPath* encapsulates a class refinement Vertex (Jak keyword refines, Line 7) and executes this refinement on the *ShortestPath* input program. The class refinement Vertex encapsulates the class members predecessor and dweight and adds a copy of these members to the Vertex class of the *ShortestPath*

[3] The term *refinement* is used ambiguously in literature [Bat04]. On the one hand, *refinement* describes the reduction of an abstraction level of a program description by adding details – ultimately such refinements generate code [Big98, Big04, Dij69, WB95]. On the other hand, *refinement* in FOP-related literature describes the synthesis of pieces of code without altering the code's abstraction level [BDN05, BLO07, BSR04, GA07]. We use the FOP-literature interpretation in this thesis because we exemplify our concepts on top of FOP.

Feature module *Directed*

```
1  public class Vertex {
2   public String name;
3   public void display(){
4    ...
5    System.out.println();
6   } }
```

Feature module *ShortestPath*

```
7  public refines class Vertex {
8   private String predecessor;
9   private int dweight;
10  public void display() {
11      System.out.print( "Pred " + predecessor +
           " DWeight " + dweight + " " );
12      Super.display();
13   } }
```

(a) Feature modules Directed and ShortestPath.

```
public class Vertex {
 public String name;
 private String predecessor;
 private int dweight;
 public void display() {
     System.out.print( "Pred " + predecessor +
        " DWeight " + dweight + " " );
     ...
     System.out.println();
  } }
```

(b) Generated program of the feature modules of Fig. 2.5a.

Figure 2.5.: *Feature modules and the program they generate.*

input program. The class refinement further encapsulates a method refinement Vertex.display()::void (Lines 10-13) and executes this method refinement on the equally named Vertex.display()::void method of its input program. The method refinement Vertex.display()::void of *ShortestPath* replaces the method Vertex.display()::void in the *ShortestPath* input program by a copy of the method refinement; but, as the method refinement calls the refined method (keyword Super, Line 12), the input program's method is integrated and essentially extended instead of replaced. We show the result of executing *Directed* and *ShortestPath* in Figure 2.5b.

Bounded quantification is a guideline to reduce the complexity of transformation-based systems such as FOP programs [AKB08, LHB05, Par78]. According to this guideline, the code generated by a feature module should not reference pieces of code that do not exist in the input program of this feature module and are not added by this feature module; that is, a feature module should not generate code with dangling references.

Feature modules (so far) only create code in input programs but never delete code therein. Thus, researchers called feature modules of FOP *monotonic* [ALMK10, BB08].[4] Feature modules (so far) enumerate every piece of code, which they transform in the input program; that is, by analyzing a feature module, one knows *all* the pieces of code the module transforms. For example, by analyzing feature module ShortestPath of Figure 2.5a, we know that this module transforms the pieces of code Vertex and Vertex.display()::void of its input programs. Thus, we call feature modules of Jak *enumerative*. For brevity, we call monotonic, enumerative feature modules as

[4]Monotonicity can hold with respect to different properties of programs [Bax90]. Feature modules are monotonic with respect to code structure.

```
public interface VertexI{
  public void display();
}

//wrapper
public class ADT implements VertexI{
 private Vertex wrappee;
 public void display(){
  wrappee.display();
 } }
```

```
//wrappee
public class Vertex {
 public String name;
 public void display() {
  ...
  System.out.println();
 } }
```

Figure 2.6.: *Program of Fig. 2.3 after executing the refactoring "Rename method ADT.show()::void into display".*

in Jak *Jak-like feature modules*. In this thesis, we denote the execution of Jak-like feature modules by the operator •; for example, we denote the consecutive execution of *Directed* and *ShortestPath* by (*ShortestPath* • *Directed*).

2.3. Refactoring

Decomposing code into modules helps to extend code [MEDJ05, Opd92], helps to understand code [Fow99, JF88, Opd92], and helps to reuse code [Opd92]. However, there generally is no best way for how to decompose code into modules [TOHS99, Tor04]. When programmers decomposed code into modules but later discover a better decomposition approach or new decomposition requirements, then these programmers can execute refactorings [Fow99]. A refactoring is a program transformation which alters the structure of programs but not their functionality [Opd92]. Researchers catalogued refactorings [Amb03, Fow99, Li06, MF05, Opd92, Rob99] but insist that an infinite number of refactorings exist [KK04, MEDJ05].

A refactoring commonly is defined as a template that accepts parameters before it can be executed [MEDJ05, RBJ97, VEd06]. Parameters of a refactoring commonly are scoped names which reference the pieces of code to transform. For example, a refactoring that renames a method ADT.show()::void into display (Rename-Method refactoring [Fow99]) accepts two parameters: The scoped name of the method to rename (ADT.show()::void) and the name of that method after renaming (display). In the remaining thesis, we call a refactoring which accepts parameters *refactoring type*.

In order to execute the above Rename-Method refactoring ("Rename method ADT.show()::void into display"), we must execute three predefined actions: We must rename the method ADT.show()::void itself into display; we must rename all methods which override ADT.show()::void or which are overridden by this method; we must update all calls to any renamed method. We show the result of executing the discussed Rename-Method refactoring on the code of Figure 2.3 in Figure 2.6 and underline affected code.

Refactorings do not alter the functionality of their input program when this input program meets *preconditions* [Fow99, MEDJ05, Opd92, Rob99]. The above refactoring "Rename method ADT.show()::void into display" does not alter the functionality when a method show exists in the class ADT of its input program and when no parameterless method display exists in the class ADT of its input program. If ADT.show()::void does not exist, the refactoring fails as there is no method to rename; if an ADT method display without parameters exists, the refactoring fails as the refactoring cannot create a second method display without parameters in class ADT of the input program (this creation would be an error according to most programming languages [GJSB05, Gro04, Li06, MEDJ05, Str91]). Of course, functionality is only guaranteed to be maintained if all code that uses a piece of code is transformed; that is, to refactor a module to expose a different API, guarantees that the functionality of the module is maintained but does not guarantee that the functionality of a program which uses the module is maintained in case it is not updated.

Commonly, refactoring engines execute refactorings in two consecutive phases [Li06]. In a verification phase, they verify that preconditions are met by the input program. In a transformation phase, they transform the input program and generate a program as the refactoring result.

Name capture is an error in refactoring that occurs when method calls or class references reference a different method or class after the refactoring than before – this could change functionality [Li06, MEDJ05, Opd92, SLMD96]. For example, name capture might occur during a Rename-Method refactoring: When a method is renamed and overrides a second method after the refactoring, which it did not override before, then the renamed method captures the name of the second method.

Refactorings exist for artifacts of a number of languages such as OOP languages [Fow99] and SPL languages [CDCv03, KKB07, LBL06, TBD06]. Refactorings for artifacts of OOP languages (named *object-oriented refactorings*) transform OOP artifacts (packages and classes) into OOP artifacts. Refactorings for artifacts of FOP languages (named *feature-oriented refactorings*) transform FOP artifacts (feature modules) into FOP artifacts *or* OOP artifacts into FOP artifacts (e.g., feature-oriented refactorings can decompose OOP artifacts into refinements of feature modules). Thus, an object-oriented refactoring in Figure 2.5b would be to rename class Vertex into ADT; in contrast, a feature-oriented refactoring in Figure 2.5b would be to separate the code of feature *ShortestPath* into a distinct feature module (cf. Fig. 2.5a). In this thesis, we focus on object-oriented refactorings and integrate them with FOP.

2.4. Summary

In this chapter, we introduced the basic techniques that we extend and integrate later. We introduced the concepts of modular programming and SPLs and discussed the implementation approaches of OOP and FOP, respectively. Finally, we reviewed refactoring as a technique to alter the structure of programs. In this thesis, we will integrate SPL techniques with refactorings – our subject is to configure the structure of modules.

3. The Dilemma of Module Scalability

Chapter 3 shares material with [KBA09, KSA10].

On the one hand, the separation of code of multiple, independent features into a module each is beneficial (cf. Sec. 2.1, p. 5); but on the other hand, exactly this separation is difficult: Programmers can separate their code with respect to mainly one feature in general but a program commonly has multiple features and all can impose different, meaningful separations [OT00, TOHS99, Tor04] – as a result, it is difficult to choose the best separation of code into modules from the beginning. With OOP for example, programmers can separate their code into classes with respect to data-structure features – each separated class then encapsulates the implementation of a data structure; programmers, however, can also separate the same program into classes with respect to functionality features – each separated class then encapsulates a piece of functionality. Decomposing OOP programs with respect to data structures and functionality at the same time equally is not possible [Tor04].

In this chapter, we show that FOP techniques allow programmers to decompose code with respect to multiple features pretty well; further, we lead to the well-known *dilemma of module scalability*, which we mitigate with the techniques in this thesis. The dilemma will state that a programmer cannot implement a module which is full of functionality and is reusable in a number of environments at the same time. In Section 3.1, we briefly report on preliminary studies which led us to this dilemma. In Section 3.2, we describe the dilemma and review its descriptions in literature. In Section 3.3, we describe related work that tackles the dilemma. In Section 3.4, we summarize the dilemma and the approaches to solve it, and define our goals.

3.1. Motivating Studies

In mainly two studies, which analyzed different programming paradigms and which followed different goals (details are beyond the scope of this thesis), we observed a common problem. In an introductory study [KARL08], we compared techniques of FOP with aspect-oriented programming – aspect-oriented programming is a modularization technique, which can be used to implement SPLs and which gained much attention lately [CRB04, GJ05, GSF$^+$05, KAB07, KPRS01, LHBC05, XMEH04, ZJ03].[1] We compared FOP and aspect-oriented programming qualitatively by analyzing programs that have been implemented with either technique (we translated

[1]Aspect-oriented programming extends modularization techniques by means of aspects [KHH$^+$01, KLM$^+$97, Lad03]. An aspect mainly adds code to a base program [MKD03].

```
refines class BinaryTreeLeaf implements VisitableNode{
 public void accept(Visitor visitor){
  visitor.visitLeaf(this);
 } }
```

(a) Available class refinement.

```
refines class LineConnection{
 public void visit(FigureVisitor visitor){
  visitor.visitFigure(this);
 } }
```

(b) Required class refinement.

Figure 3.1.: *Personal experience with the module-scalability dilemma [from KSA10].*

programs of aspect-oriented programming into FOP counterparts before in different variants [KRAL07]).

For all programs, we analyzed to what extend code of a feature could be reused as independent module across *independently* developed programs and across *dependently* developed programs (e.g., across products of an SPL). Furthermore, we analyzed to what extend we could group code of features into modules that are valuable to reuse. In summary, we observed that FOP's feature modules perform pretty well with respect to reusing modules across dependently developed programs and with respect to the grouping of code of features. However, we also observed that feature modules may hardly be reused across independently developed programs as the references of feature modules to pieces of code, which they transform, are fixed – this limits feature modules to be reused only with programs that actually provide the referenced pieces of code. So far, we did not attach much attention to the lack of reusability of feature modules across independently developed programs.

In a follow-up study, we applied FOP techniques to large-scale Java programs (e.g., to the user-interface framework JHotDraw[2] with 30K lines of code, and the embedded-database engine Berkeley DB[3] with 90K lines of code) [KSA10]. We mainly executed feature-oriented refactorings on the studied programs to separate code of features into feature modules.

In the follow-up study, we again faced the lack of reusability of feature modules across independently developed programs (we did not attach much attention to this problem in our introductory study before): In our follow-up study, we had to implement a class refinement which creates a single method; specifically, a method with one parameter, no return value, and a method call on the single parameter variable with the pseudo variable this as the parameter value. Although such class refinement existed in the code of our introductory study (cf. Fig. 3.1a), we could not reuse this refinement because the names it provided and used for classes and methods differed from what we desired now. Now, we desired a refinement which transforms a class LineConnection instead of BinaryTreeLeaf and which creates a method visit instead of accept; the visit method should now accept a FigureVisitor object as parameter instead of a Visitor object and should call method visitFigure instead of visitLeaf. As we could not reuse the code, we reimplemented its functionality and show our new module in Figure 3.1b; that is, we introduced a code clone (code clones are considered

[2]`http://sourceforge.net/projects/jhotdraw/` (accessed: January 7,2010)
[3]`http://www.oracle.com/database/berkeley-db/je/` (accessed: January 7,2010)

harmful [RC07]).[4]

We were astonished but learned a lot from the reuse failures in both of our studies. We aimed to reuse a module, which implements a certain piece of functionality that we desired for our new program, but could not do so. We learned that the code structure of a module can prevent the module's reuse.

3.2. Defining the Dilemma of Module Scalability

Simple name conflicts hindered us to reuse the code of our own module. This happened, although the code to reuse implemented exactly the functionality we desired. We analyzed the problem and recognized on the basis of literature [BCS00, Big98, CHSV97, CL01, Her08, HM07, Mey97, OT00, TOHS99] that a module must follow conflicting goals:

- *The more functionality, the better the module:* A description of the functionality of a module tells a reusing programmer *what* has been implemented in this module. When a module provides a lot of functionality, this functionality more likely is part of the functionality, which a programmer is about to implement for a new program. If the module functionality covers the required functionality, then the programmer can save the implementation effort for this functionality by reusing the module. The more functionality a programmer can reuse from a module, the less functionality he/she must implement on his/her own. However, only the large-scale modules foster the fast and efficient implementation of new programs – orchestrating a high number of small-scale modules to provide a certain piece of functionality is not as beneficial.
- *The fewer decisions exposed regarding structure, the better the module:* A description of the decisions regarding the structure of a module tells a reusing programmer *how* functionality has been implemented in this module. For example, decisions regarding a module's structure include the allocation and the name of a piece of code which implements certain functionality. When a module exposes few decisions regarding its structure (e.g., few names) it is unlikely that these decisions conflict with decisions that were already made regarding the structure of an environment; as a result, the module can be reused in many environments. By symmetry, if modules expose high numbers of decisions regarding their structure, one of these decisions more likely conflicts with the decisions already made for reusing environments, and then the module cannot be integrated with these environments as is.

The two goals above conflict because to implement the functionality of a module, programmers *must* decide on how to implement it; that is, the programmers *must*

[4]As we reimplemented the module's functionality, we skipped the API artifact VisitableNode because we did not need it in this follow-up study. Otherwise, the code of Fig. 3.1a and Fig. 3.1b would have been even more similar.

make a number of decisions regarding a module's structure. This dilemma of module scalability is also inevitably connected to APIs: The goal to provide a lot of functionality calls for a rich API to access every piece of functionality on its own; the goal to expose few decisions regarding structure, in contrast, calls for a narrow API with few names and few definitions of code allocations [BCK06, Mey97, PBvdL05]. In the remaining thesis, we (as others did before [FCDR95, GJSB05, MSL00]) refer to each name conflict and each other conflict in structure as an *incompatibility.*

The module-scalability dilemma is prominent in literature and has multiple names:

- Bertrand Meyer characterizes the *reuse-redo dilemma* as "the central problem of software reuse" [Mey97]. Meyer describes that programmers must either reuse a "frozen" (unchangeable) module with its entire functionality and its entire set of decisions regarding its structure as is or must redo the implementation of the module functionality. Meyer insists that unchangeable modules are not very reusable in practical software development and concludes that modules must be adaptable. Meyer further concludes that modules must provide a suitable functionality together with a suitable structure (API) to be reusable [Mey97].

- Ted J. Biggerstaff reports that the *vertical-horizontal scaling dilemma* is a fundamental problem of modular programming [Big98]. Biggerstaff describes that programmers gain highest reuse payoffs when they reuse large-scale modules with a lot of functionality included. But Biggerstaff also describes large-scale modules as being domain-specific and reusable in fewer environments than modules with less functionality. Biggerstaff argues that a programmer has less effort to reuse one large-scale module than to reuse a high number of small-scale modules which together provide the large-scale module's functionality. He concludes that problems in a module's structure frequently prevent to reuse the module.

- Peri Tarr et al. characterize the problem of a *dominant decomposition* as a major problem with respect to software reuse [OT00, TOHS99]. Tarr et al. describe that modules should be large-scale. But Tarr et al. also describe that large-scale modules expose many decisions regarding their structure and, thus, are reusable in fewer environments than small-scale modules. They showed that these decisions exist mainly because programmers must select one dominant feature (they call it *concern*) to decompose their code into modules. Tarr et al. conclude that large-scale modules commonly lack reusability. They further conclude that programmers should allow users to adapt a module without changing the module code.

- Stephan Herrmann et al. describe the *encapsulation-adaptation dilemma* [Her08, HM07]. Herrmann et al. show that encapsulation (i.e., a fixed structure (API)) is a "key concept for modular software designs" and among other things is important for reusing a module [Her08]. But Herrmann et al. also show that modules rarely fit the needs of reusing environments exactly – they conclude that modules should be adaptable.

- Wim Codenie et al. characterize the problem of *overfeaturing* for object-oriented frameworks [BCS00, CHSV97, CL01].[5] Codenie et al. describe that programmers commonly increase the functionality of a framework (a) to reduce the effort of implementing the classes that reuse the framework and (b) to avoid repeated implementation of a feature across these classes [BCS00, CHSV97, CL01]. However, they also describe that frameworks with too much functionality (i.e., overfeatured frameworks) are less reusable as they are too domain-specific [BCS00, CHSV97, CL01].

The researchers cited above show that reuse of modules frequently fails because the modules provide unsuitable functionality, unsuitable structure (set of decisions regarding structure), or both. These researchers argue that modules cannot scale with respect to functionality without causing incompatibilities with environments.[6] In this thesis, we tackle the module-scalability dilemma; we reuse an approach, which allows programmers to scale the functionality of modules, and extend it to alter the structure of modules.

3.3. Related Work on Module Integration

From the beginning of modular programming, modules had to be integrated with environments (i.e., with code outside the module). Approaches used for module integration cover wrappers, mediators, generic programming, meta-programming (we separately discuss the special case of refactoring-like meta-programming), SPLs, and integrated development environments. We now discuss these approaches with respect to configuring the functionality *and* the structure of a module. Thereby, users should be able to define configurations without knowing anything about the module's implementation. That is, the configuration of the structure and the functionality of a module should be based on the concept of features. Furthermore, decisions regarding the functionality of a module should not affect decisions regarding the structure of this module, and vice versa. We call this an *integrated, feature-driven configuration* of a module's functionality and structure.

Wrappers. Wrappers can virtually alter the API of wrappees (cf. Sec. 2.1). If the wrappee is a class, the wrapper is called *class wrapper*; if the wrappee is an object, the wrapper is called *object wrapper*.

Class wrappers are subclasses of wrappees (wrappees here are classes); that is, wrappers inherit members of their wrappees and define forward methods [GHJV95]. The forward methods call the inherited wrappee methods. If a wrapper method has a different name than the wrappee method which it calls, this wrapper method provides an additional name to access the wrappee method. As class wrappers are subclasses of their wrappees, objects of class wrappers can be used where objects of

[5] Frameworks are one approach to reuse code in OOP (cf. Sec. 2.1, p. 9).
[6] Some of these researchers present approaches, too. We review these approaches later.

wrappees have been used before. In the end, objects of a class wrapper provide the functionality of objects of the respective wrappee but provide additional names to access this functionality. Note that wrapper methods can perform additional actions before and after forwarding to wrappee methods. In Figure 3.2a, class ADT is a class wrapper of class Vertex (i.e., ADT inherits from Vertex; Line 1) and provides a method show to access method Vertex.display()::void.

Object wrappers are objects of which each has a field to reference a wrappee (wrappees here are objects) [GHJV95]. Object wrappers commonly provide methods that forward to methods of wrappees. If a wrapper method has a different name than the wrappee method which it calls, this wrapper method provides an alternative name to access the wrappee method – however, using the wrapper, the wrappee method is not directly accessible with its own name. If a wrapper does not provide a forward method for a wrappee method, the wrappee method is not directly accessible using the wrapper. A wrapper and its wrappee can only be assigned to the same variables when these variables have types of common superclasses of the wrapper's class *and* the wrappee's class. Note that wrapper methods again can perform additional actions before and after forwarding to wrappee methods. Object wrappers allow programmers to replace modules with modules that are written in different languages and different programming paradigms; wrappers then include the code to translate between the languages [Cle09]. In Figure 3.2b, class ADT is a class of object wrappers that wrap objects of class Vertex (i.e., ADT references Vertex objects with a member variable; Line 2) and provides a method show to access method Vertex.display()::void.

Wrappers share problems (i.e., they might increase complexity). Wrapper classes (i.e., class wrappers or classes of object wrappers) do not remove wrappee classes (i.e., wrappees of class wrappers or classes of wrappees of object wrappers); decisions regarding the structure of wrappee classes (e.g., class names) thus can still prevent a module's reuse when name conflicts occur. Wrapper classes may require environments to change because environments must use wrapper classes instead or in addition to wrappee classes (e.g., to create wrapper objects) [Höl93].[7] Wrapper classes add code to modules and thus increase the effort to maintain these modules [Bos98, Höl93, MB97]. Wrapper classes have different positions in inheritance hierarchies than wrappee classes; these positions cause complex redundant class hierarchies of wrappers [Höl93]. For example, in Figures 3.2c & 3.2d, we show wrapper classes for wrappee classes that are part of an inheritance hierarchy; the hierarchy of the wrapper classes ADT and SpADT is redundant to the hierarchy of the wrappee classes Vertex and SpVertex. When different decisions regarding the structure of a wrappee should be configurable *independently*, we must introduce numbers of wrappers (with numbers of cloned methods) or we must introduce complex class hierarchies and forward-method chains (e.g., using the object-oriented design pattern Chain of Responsibility [GHJV95]). As example, suppose that a class has two

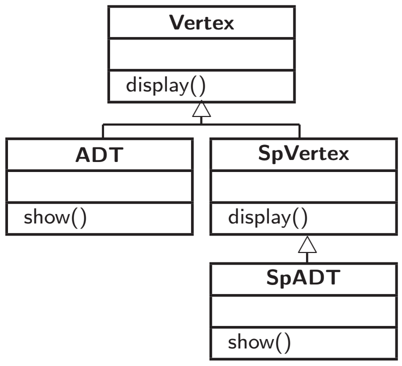
[7]Commonly, additional object-oriented design patterns are used to prepare environments for wrapper changes (e.g., Factory Method) – but this adds complex code [Höl93].

```
public class ADT extends Vertex{
  public void show(){
    super.display();
  } }
```

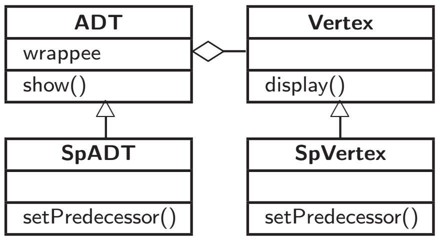
(a) Class ADT *as a class wrapper for class* Vertex *of Fig. 2.5b.*

```
public class ADT{
  Vertex wrappee;
  public void show(){
    wrappee.display();
  } }
```

(b) Class ADT *as an object wrapper for class* Vertex *of Fig. 2.5b.*



(c) Multiple class wrappers for Vertex *classes.*



(d) Multiple object wrappers for Vertex *classes.*

Figure 3.2.: *Object wrappers and class wrappers.*

methods display and setPredecessor; to provide new names for both methods independently, we need three wrappers: The first wrapper provides a new name for setPredecessor, the second wrapper provides a new name for display, and the third wrapper provides new names for both. Wrappers can only wrap code and alter code, which they can reference, but not code they cannot reference; for example, wrappers can neither wrap nor alter blocks of statements in the middle of wrappee methods. The implementation of wrapper classes cannot reuse wrappee members, which are modified with private, but must reimplement them and possibly every method calling them.

Class wrappers have unique problems. Class wrappers clone code when different wrappee classes should be wrapped homogeneously; in languages with multiple-inheritance support (i.e., in which one class can inherit members from different classes) such as C++, one wrapper class can wrap different wrappees but need complex and fragile precedence declarations to avoid ambiguities [Str91] – these precedence declarations however are fixed and cannot vary when wrappees vary. Redundant class hierarchies of class wrappers might include code clones. For example, to change the APIs of classes Vertex and SpVertex in Figure 3.2c, a programmer must implement a subclass for Vertex (ADT in Fig. 3.2c) and a subclass for SpVertex (SpADT in Fig. 3.2c); the wrapper classes then clone members such as show. Interestingly, in Figure 3.2c, the class wrapper SpADT does not inherit from the class wrapper ADT and so SpADT objects cannot be assigned to variables that have ADT as their type.

Object wrappers have unique problems, too. Object wrappers impair performance because creating respective (wrapper) objects takes time and because these objects' forward calls take time [Höl93]. Object wrappers impair footprint because they add (forward) methods [Höl93]. Object wrappers increase the development effort because programmers must implement a wrapper method for *every* wrappee method that should be accessible when the wrapper is used (even if the wrappee method should not change) [Bos98, MB97]. Object wrappers increase the complexity of code because they add objects, which must be managed at runtime [BCL10, DNMJ08, HA09, Höl93, Kni99, MHM09, SR02]. As a result, a wrappee object is not equal to its unwrapped self and a wrappee object can incorrectly execute a wrapped method without executing the wrapping code; for example, in Figure 3.2d, a Vertex method might incorrectly call other Vertex methods without executing their ADT wrapper code. Object wrappers also increase complexity because they are located in different positions in inheritance hierarchies than wrappee classes [Höl93]; wrapper objects thus cannot always be used where wrappee objects were used before. Object wrappers become especially complex when environments expect code in different positions than it is provided inside the module (as a result, simple forward methods do no longer suffice); this culminates in wrappers that reimplement functionality of wrappees [BCLvdS10].[8] The implementation of object wrappers cannot reuse wrappee members, which are modified with protected, but must reimplement them and possibly each method calling them.

Tools can generate wrapper classes but these tools are limited. First, metaprograms can be triggered in the code of wrapper and wrappee classes; for example, macros can define name mappings from which forward methods are generated or macros can introduce language constructs which replace wrapper classes (we discuss these and other meta-programming approaches on page 26 in more detail). Second, tools can create wrapper classes from a recorded or inferred sequence of program transformations which transformed the API of a program into the API of a new revision or of a different version [DNMJ08, ŞR07]; these tools however need a sequence of refactorings or the complete code of both revisions/versions. That is, with these approaches we cannot implement empty wrappers for a program and expect these tools to complete the wrappers with respect to an incompatible module. Finally, tools can generate wrappers, when modules are replaced by modules that expose equal functionality but that are written in a different language [Cle09].

Special wrapper-like approaches or approaches that can elegantly add wrapper code [HO93, JMS07, LLM99, LO06, MO02, MSL00, OT00, OT01, SB98, TOHS99, WT09] among others allow programmers to add members to classes (e.g., mixins

[8] Others [BCLvdS10] observed that a major part of incompatibilities (API mismatches) at method level are simple: Methods map one-to-one to methods of the reused module; this mapping for example requires (un)wrapping, altering of argument positions, or replacing of this return values.

allow to add forward methods or methods with certain names to classes) or to synthesize methods. Individual approaches additionally support to rebind names that a module expects from its environment [LLM99] but mechanisms beyond the adaptation of names were not discussed for these individual approaches; interestingly, the adapted module of the last approach [LLM99] adapts the module's environment (e.g., by replacing methods).

The integrated, feature-driven configuration of a module's functionality and structure has not been discussed for the above approaches. That is, in general, programmers must verify *on their own* that their wrappers maintain the functionality of wrappees (if intended).

Mediators. A module may instruct a mediator program or middleware to interact with a second module (e.g., to call a method on this second module or to pass values of variables [GHJV95, Gro04]). The mediator program then is responsible to locate the requested module, to translate and perform the instruction, and to return the instruction result [Gro04]. With mediator programs, modules need not comply any more with decisions regarding the structure of other modules but only with decisions regarding the structure of the mediator program [GHJV95, Gro04]. The mediator structure (API) that modules must comply with can be overly complex and inconsistent [Hen08]. Mediator programs provide no concepts to *configure* the structure or functionality of a single module.

Generic programming. A template program of generic programming commonly accepts as a parameter a class name that supersedes placeholders inside the template program [CE99a, CE00, Gho04, GJSB05, Str91]. This way, template programs can be adapted to environments at the time the template program is used (i.e., after its implementation). For example, an environment can call a template program and pass a class name as a parameter; this class name supersedes placeholders inside the template program and as a result, the template program can manipulate objects of the passed class' type (possibly defined in environments). Finally, an environment can *specialize* a template program; the template parameter then is used to select between alternative implementations inside the template program [CE99b, CE00]. However, in some approaches, programmers must clone the template class to alter the name of a method; to alter the name of a class that is not nested in other classes is usually not possible at all. Furthermore, template programs can only avoid incompatibilities with their environments when the conflicting decision regarding their structure has been prepared for alteration with a template parameter. Mixins are special template classes and are evaluated as special wrapper-like approaches on page 25. In single approaches, templates can be generated from stand-alone programs; template parameters can then even rename classes that are not nested in other classes [KR05]. All these approaches do not support to configure the structure *and* the functionality of a single module, especially not an integrated, feature-driven configuration of a single module's functionality and structure.

Meta-programming. A high number of approaches allows meta-programs to transform input programs and/or to generate code based on input parameters (e.g., to remove incompatibilities between a module and its environment) [Aßm98, Bax90, Bax92, Big04, BKVV08, BPM04, BTF05, CE99a, CM07, Gog96, HZS07, JS03, Nov95, Nov97, SdML04, TC98, TCKI00]. Meta-programs may alter the structure and the functionality of programs; that is, commonly, meta-programs are little or not restricted in what they change. For that, in many approaches, programmers must verify *on their own* that a meta-program does not alter an input program's functionality (if intended). In some approaches, programmers even must verify *on their own* that the generated program compiles; only some approaches incorporate tools to verify that every generated program compiles [HZS05].

Macros transform pieces of code in an input program, which have been annotated or which have an extended syntax [BCVM02, Bos98, HS08, HZS07, KKA10, ML98, Str91, VRB00] (some macros describe name mappings and are used to generate wrapper mechanisms [ML98]). Annotations can be comment-like entities or pieces of code. An annotation can be replaced by macros with code, or an annotation can trigger transformations related to that annotation. For example, a method's annotation can trigger a transformation on the annotated method or on related code [BCVM02]. In general, macros are not restricted in what code they replace and so programmers must verify *on their own* that a macro does not alter a program's functionality (if intended). In some approaches, programmers even must verify *on their own* that all generated programs compile (e.g., for C++ macros [Str91]).[9] Macros are controversial and should be avoided if possible [Str91].

Refactoring-like meta-programming. Some meta-programming approaches as well as some macro-like approaches are restricted to restructure code [Int07, Läm02, TCKI00, Tho05, VEd06] or have been used to restructure code [EH07, LGS09, PRT08]; thus, these approaches can guarantee to generate programs that compile and that do not differ in functionality. However, all these approaches have not been analyzed with respect to configuring the structure *and* the functionality of a single module (they also have not been utilized for this), especially not with respect to an integrated, feature-driven configuration of a single module's functionality and structure. Individual meta-programming approaches allow arbitrary changes to integrate a module; changes that are refactorings (Rename Field, Rename Method, and Rename Class) and changes that add, remove, or alter code in an unrestricted way [KH98] – these transformations have not been modeled as configurable features of individual modules.

Some languages allow programmers to define additional names for named pieces of code [Jon03, MHQB05, Ode10, Str91]. These approaches do not remove names of a module (names which may be the reason for an incompatibility) and do not allow programmers to configure the structure of a module beyond names. Finally,

[9] C++ macros ignore the syntax of C++ as well as the semantics of C++ mechanisms such as the semantics of access modifiers; thus, macros may generate programs which do not compile [Str91].

they do not support an integrated, feature-driven configuration of a single module's functionality and structure.

Software product lines. SPLs can be used to implement configurable modules [BCS00, BSST93, CE99a, CE99b]. Module users here select features that represent pieces of functionality. An SPL product then includes code which implements the user-selected features but generally does not include the code of those features the user did not select. In this respect, the number of decisions regarding structure exposed by the SPL product is smaller than the number of decisions exposed by a fully-fletched program.

Techniques to configure SPL products with respect to structure currently impose problems: To alter an SPL product's structure, first, SPL users may alter their feature selection [KRT97] – this, however, results in a product the users might not be interested in. Second, users may choose from alternative pieces of code that implement the same features but with a different structure – however, to implement alternatives is costly and to maintain them is error-prone [Fow99]; alternative implementations often either involve code clones or a complex structure.

Code that has been assigned to implement a feature may include wrappers; as a result, a module of an SPL product that was generated with the wrapper feature can be accessed using these wrappers [BCK06]. However, in addition to the problems discussed for wrappers above, wrapper classes in SPLs might need to be configured just as their according wrappee classes are – this increases complexity. For example, suppose that ADT is a class of object wrappers for Vertex objects (not shown), then ADT *must* provide a method that forwards to Vertex.setPredecessor()::void if this wrappee method exists in a product; however, ADT *must not* provide this forward method otherwise to avoid dangling references in the wrapper and thus to avoid compiler errors. If ADT is a class wrapper for Vertex and encapsulates a method that forwards to Vertex.display()::void, then ADT must change if Vertex.display()::void does not exist in a product; if ADT would not be changed in this case, dangling references and compiler errors would occur.

Special concepts used for SPLs can alter the functionality and the structure of a program to some extend (e.g., aspect-oriented programming can connect classes via inheritance) [ALS08, ZGJ05]. Other concepts use template programs to configure types, which variables of a module have, independently from this module's functionality [AKL06]. These concepts do not help when names of classes, which are not nested in other classes, conflict with an environment, or when the allocations of pieces of code conflict. Finally, macros and preprocessor directives can be used to alter a module's structure [Nat06, ZJ04] – but this either is complex or involves code clones. For example, to alter the name of a package involves fine-grained changes to numerous pieces of code throughout the program; changes that are difficult to implement without error [KAK08].

Table 3.1.: *Assessment of related work on module integration.*

Criteria	**Wrappers**	**Mediators**	**GP**$^\alpha$	**MP**$^\beta$	**RLMP**$^\chi$	**SPLs**	**R-IDE**$^\delta$
Functionality configuration	$\odot$	$\ominus$	$\odot$	$\oplus$	$\ominus$	$\oplus$	$\ominus$
Structure configuration	$\oplus$	$\ominus$	$\odot$	$\odot$	$\oplus$	$\odot$	$\oplus$
Configuration independence & integration	$\odot$	$\ominus$	$\oplus$	$\odot$	$\oplus$	$\odot$	$\oplus$

$^\alpha$Generic programming; $^\beta$meta-programming; $^\chi$refactoring-like meta-programming; $^\delta$refactoring in integrated development environment; $\oplus$good support; $\odot$problematic support;$\ominus$bad/no support

Refactoring with integrated development environments. A number of integrated development environments allow programmers to restructure stand-alone programs with refactorings (e.g., to remove incompatibilities) [FKK07, FTK04, Li06, RBJ97] or with refactoring-like transformations [Smi90, Smi91]. Programmers define the refactoring parameters by selecting code in the program and completing dialog boxes of the integrated development environment. However, users who want to refactor programs with integrated development environments, must *know the program* in order to define where to apply a refactoring and must *know the refactorings* to ensure that the refactorings generate the desired structure (i.e., they do not allow to configure using features). Some approaches even allow programmers to execute individual refactorings on SPLs [Vit03]. However, structure can still not be configured using features.

3.4. Summary and Goals

In this chapter, we reviewed the well-known dilemma of module scalability. We reported briefly on preliminary studies and described how we experienced the dilemma. Accordingly, we cannot scale the functionality of a module without reducing the number of environments this module can be integrated and reused with. We reviewed descriptions of this dilemma in literature. Finally, we assessed to what extent and how existing techniques can help to mitigate the dilemma; that is, how they support an integrated, feature-driven configuration of a module's functionality and structure. We summarize our assessment in Table 3.1. We conclude that no approach supports our aim satisfactorily, i.e., no approach supports an integrated, feature-driven configuration of a module's functionality and structure.

In this thesis, we aim at an approach that supports an *integrated, feature-driven configuration* of a single module's functionality and structure. That is, we aim at an approach that supports programmers to configure the functionality *and* the structure of a module. Thereby, programmers should be able to define configurations without knowing anything about the module's implementation. To achieve this, we aim at techniques that allow programmers to configure the structure of a module based on the concept of features (techniques that allow programmers to configure functionality based on features already exist). We aim at an approach which ensures that decisions regarding the functionality of a module do not affect decisions regarding the structure of this module, and vice versa. We will discuss such an approach in Chapter 4. We target at automated consistency checks between the code of a configurable module and the feature model of such a module (cf. Chap. 5). We want to understand the implications of our approach in detail and so we will analyze and evaluate this approach with respect to the correction of modules, with respect to the configuration-process implications, and with respect to covering multi-language programs (cf. Chap. 6).

4. Refactoring Feature Modules

Chapter 4 shares material with [KBA08, KBA09, KKAS11, SKAP10].

In this chapter, we propose *refactoring feature modules (RFMs)* to mitigate the module-scalability dilemma, as described in Chapter 3.[1] RFMs are special feature modules of an SPL; they encapsulate refactorings instead of classes and class refinements. RFMs and groups of RFMs correspond to features in the feature model; features that do not differ from features which Jak-like feature modules correspond to. RFMs allow users to configure a program or module with respect to its structure.

4.1. Concept

RFMs are special feature modules, which do neither encapsulate classes nor class refinements; instead, each RFM encapsulates one refactoring unit. A *refactoring unit* is a class-like entity which describes parameters for a specific refactoring type.[2] In particular, refactoring units first declare which refactoring type they implement (e.g., the refactoring unit MyRename of Figure 4.1 starts with MyRename implements RenameMethodRefactoring to declare that it implements a Rename-Method refactoring; Line 1). The tool, which we prototypically implemented to execute Jak-like feature modules and RFMs (*RFM composer tool* or *composer tool* for short), knows the API artifact of every refactoring type it supports and expects the units accordingly to provide certain methods; each of these methods returns this unit's value for a specific parameter of the implemented refactoring type.[3] When a refactoring unit references a refactoring type and provides all the parameters of this type with according methods, then this unit is executable; when a refactoring unit references a refactoring type but does not provide all the parameters of this type with according methods, then the refactoring is not completely specified and the composer tool reports an error. With respect to the refactoring unit of Figure 4.1, the composer tool knows the RenameMethodRefactoring API artifact and thus enforces MyRename to provide the methods getOldMethod (to return the refactoring parameter of the old

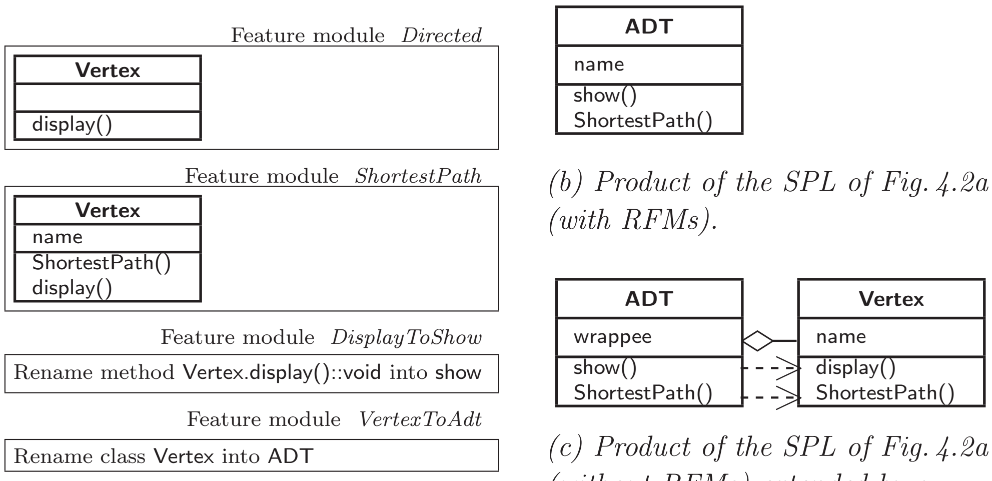
[1] The term *module* commonly is connected to providing functionality (cf. Sec. 2.1) but RFMs will not provide functionality on their own. In the best case, they separate pieces of functionality from other modules and make these pieces accessible. However, RFMs correspond to features of SPLs and make SPL products provide features (regarding structure). To emphasize the correspondence to Jak-like feature modules, which also make SPL products provide features and which also are program transformations used in SPLs, we stick with the term *module* for RFMs.

[2] A refactoring type is a transformation template which expects parameters (cf. Sec. 2.3, p. 15).

[3] To prototypically implement the composer tool, we partly reused existing tools and modules (cf. Sec. 4.2.3).

```
refactoring MyRename implements RenameMethodRefactoring {
  String getOldMethod(){return "Vertex.display()::void";}
  String getNewMethodName(){return "show";}
}
```

Figure 4.1.: *Refactoring unit inside an RFM that renames method Vertex.display()::void into show.*



(a) SPL with RFMs.

(b) Product of the SPL of Fig. 4.2a (with RFMs).

(c) Product of the SPL of Fig. 4.2a (without RFMs) extended by a wrapper class.

Figure 4.2.: *SPL code with RFMs, and SPL products [adapted from KBA09].*

method name) and getNewMethodName (to return the refactoring parameter of the new method name) – if either method would have not been provided in MyRename, the composer tool would report an error. In Figure 4.1, MyRename provides all required methods and thus can execute and rename method Vertex.display()::void of an input program into show. Refactoring units of other refactoring types are defined analogously.

RFMs integrate with Jak-like feature modules of FOP. In Figure 4.2a, we show the Jak-like feature modules *Directed* and *ShortestPath* of our GPL running example and added to them the RFMs *DisplayToShow* and *VertexToAdt*.[4] *DisplayToShow* encapsulates the refactoring unit MyRename of Figure 4.1 (i.e., it renames method Vertex.display()::void into show). *VertexToAdt* encapsulates a refactoring unit that implements a Rename-Class refactoring to rename class Vertex into ADT. When a user selects the features *Directed*, *ShortestPath*, *DisplayToShow*, and *VertexToAdt*, then all feature modules of Figure 4.2a execute consecutively in top-down order and generate the program which we show in Figure 4.2b. Note that this program of Figure 4.2b

[4]Throughout this thesis, we name RFMs according to explanatory reasons only (i.e., as simple as possible). In practice, names for RFMs would certainly describe the domain concept of the feature they contribute.

has basically the same properties as a program generated from the Jak-like feature modules of Figure 4.2a extended by a wrapper class of Vertex, ADT (cf. Fig. 4.2c): Class Vertex provides the functionality of features *Directed* and *ShortestPath*, Vertex can be accessed with the name ADT, and method Vertex.display()::void can be accessed with name show using objects of type ADT. In contrast to the wrapper approach (cf. Fig. 4.2c), the program generated from the feature modules of Figure 4.2a (cf. Fig. 4.2b) contains no class Vertex any more because *VertexToAdt* removes Vertex (and replaces it with a class ADT).

Jak-like feature modules and RFMs map to features in feature models and are executed on behalf of the feature selection of a user. That is, users can still *configure the functionality* of an SPL product when they select features, which map to Jak-like feature modules. But users now can also *configure the structure* of an SPL product when they select features, which map to RFMs. Note that users do neither need to know about refactorings nor about implementation details in order to configure an SPL product with a feature model; users only need to know about features as domain concepts.

4.1.1. The Scope of RFMs During Program Generation

RFMs enforce the bounded-quantification guideline of Jak-like feature modules (cf. Sec. 2.2.2). That is, during every program generation, Jak-like feature modules and RFMs execute in the order defined for their features in the feature model; Jak-like feature modules and RFMs take the program, which was generated by other feature modules before, as an input, transform this program, and generate a program as an output. Thereby, RFMs might at most restructure existing references in their input program but do never add new references. As a result, if the input program of an RFM has no dangling references and the RFM succeeds then the generated program has no dangling references, either.

In Figure 4.2a, *DisplayToShow* and *VertexToAdt* are RFMs which integrate with the Jak-like feature modules of the GPL. *DisplayToShow* transforms the program generated by *ShortestPath* and generates a program which includes the method Vertex.show()::void. *VertexToAdt* transforms the program generated by *DisplayToShow* and generates the desired SPL product, finally. *DisplayToShow* (analogous to any other RFM) does not transform code that other feature modules generate after *DisplayToShow* was executed according to the feature order (e.g., *DisplayToShow* does not transform code generated by *VertexToAdt*, even if *VertexToAdt* would be a Jak-like feature module and would generate a method Vertex.display()::void).

4.1.2. RFM Refinement

RFMs are transformations but are generally not translated into artifacts of the generated program. Nevertheless, RFMs can vary across SPL products. We show now that RFMs can be valuably refined with FOP techniques for certain use cases. As a result, RFMs now can also encapsulate refinements of refactoring units; as a second

Feature module *makeCompatible*

```
 1 import java.util.Map;
 2 import java.util.HashMap;
 3 public refactoring AddParam implements AddParameterRefactoring {
 4   public String getMethodToChange(){ return "Vertex.display()::void"; }
 5   public String getFormalParameterType(){ return "String"; }
 6   public int getFormalParameterPosition(){ return 0; }
 7   public String getDefaultActualParameter(){ return "\"VERBOSE\""; }
 8   public Map getSpecificActualParameters(){
 9     Map params = new HashMap();
10     params.put("Graph.display()::void", "\"RELEASE\"");
11     return params;
12   } }
```

(a) RFM and its refactoring unit AddParam *that implements Add Parameter.*

Feature module *makeCompatibleTestcase*

```
13 public refines refactoring AddParam {
14   public Map getSpecificActualParameters() {
15     Map params = Super.getSpecificActualParameters();
16     params.put("TestCase.testDisplay()::boolean", "\"DEBUG\"");
17     return params;
18   } }
```

(b) Refinement for refactoring unit AddParam *of Fig. 4.3a.*

Figure 4.3.: *Refinement for an Add-Parameter RFM [adapted from KBA08].*

result, we can reuse individual RFMs for the generation of more SPL products.

We argue that RFM refinement is valuable, for example, to implement a set of similar Add-Parameter RFMs.[5] In Figure 4.3, we demonstrate the refinement of an Add-Parameter RFM using the RFMs *makeCompatible* and *makeCompatibleTestcase*. The RFM *makeCompatible* encapsulates a refactoring unit AddParam which implements an Add-Parameter refactoring; AddParam adds a new first parameter variable, which has the type String, to method Vertex.display()::void (Lines 4-6) of an input program; AddParam defines that method calls should pass the string VERBOSE by default as a value for the new parameter (Line 7); AddParam defines that calls in method Graph.display()::void of the input program should pass RELEASE as a value for the new parameter (Lines 8-12). The RFM *makeCompatibleTestcase* encapsulates a *refinement* of the refactoring unit AddParam; the refinement defines that the calls in method Testcase.testDisplay()::boolean extended by AddParam should pass the value DEBUG as the value for the new parameter instead of the default value (Line 16). If TestCase.testDisplay()::boolean is absent in a program, we can still execute *makeCompatible* without error (without *makeCompatibleTestcase*). We show more possible use cases of RFM refinement in Table 4.1

[5] Add Parameter is a refactoring type that adds a parameter variable to a method and updates all calls to that method to pass an additional parameter value [Fow99].

Table 4.1.: *Refactoring types for which RFM refinement could be beneficial.*

Refactoring type$^{\alpha}$	**Refinement benefit**
Add Parameter	The set of method-specific parameters could be extended
Extract Class	The set of class members to extract could be extended
Extract Superclass	The set of class members to extract could be extended
Inline Method	The set of methods which should inline a method could be extended
Pull Up Constructor Body	The set of constructors to pull up could be extended
Pull Up Field	The set of fields to pull up could be extended
Pull Up Method	The set of methods to pull up could be extended
Remove Middle Man	The set of forward methods to remove could be extended

$^{\alpha}$Refactoring types described elsewhere [Fow99]

RFM refinement could impair bounded quantification so we must intervene.[6] Currently, RFM refinements *could* make refined RFMs transform code incorrectly, which was added after the refined RFMs executed (but before the RFM refinements). We forbid this. That is, to keep complexity as low as possible, we define that RFM refinements must be immediate successors of the RFMs, which they refine, according to the feature order (e.g., in Figure 4.3, we do not allow any feature module to execute between *makeCompatible* and *makeCompatibleTestcase*).

4.2. Algebraic Properties of Refactorings Influence the RFM Tools

In the course of implementing an RFM composer tool, we analyzed algebraic properties of operations, which abstractly describe refactorings and refinements [KKAS11]; for example, we analyzed whether there are inverse operations in the domain of refactorings. We present this analysis here because it explains our implementation decisions regarding the RFM composer tool and because it can influence lines of research beyond RFMs (e.g., the implementation of refactoring tools for SPLs and the implementation of other transformation-based SPL techniques [SD10]). In this analysis, F denotes a Jak-like feature module; R denotes an RFM operation (i.e., a refactoring); $R_{\mathsf{A}\mapsto\mathsf{B}}(F_1)$ denotes a refactoring which transforms the code of the Jak-like feature module F_1 to replace a piece of code A by a piece of code B.

For an RFM composer tool, algebraic properties of refactorings and refinements are important because we identified two approaches to implement such tool (cf. Fig. 4.4). To execute the sequence $R((F_2 \bullet F_1))$ of feature modules, a composer tool can:

(a) Execute the operations of the Jak-like feature modules and RFMs (refinements and refactorings) in feature order consecutively ($R((F_2 \bullet F_1))$, left perimeter in Fig. 4.4), or can

[6]Bounded quantification is a guideline according to which a feature module should not generate code with dangling references (cf. Sec. 2.2.2, p. 14).

$$\begin{tikzcd}
& \boxed{R((F_2 \bullet F_1))} \arrow[dl, "F_3 = (F_2 \bullet F_1)"'] \arrow[dr, "{F_4 = R(F_1) \quad F_5 = R(F_2)}"] & \\
\boxed{R(F_3)} \arrow[d, "F_7 = R(F_3)"'] & & \boxed{(F_5 \bullet F_4)} \arrow[d, "F_6 = (F_5 \bullet F_4)"] \\
\boxed{F_7} & \overset{?}{=} & \boxed{F_6}
\end{tikzcd}$$

F_x	=	program
R	=	refactoring-feature module
$\bullet$	=	program composition

Figure 4.4.: *Different implementation approaches for RFM composer tools [from KKAS11].*

(b) Execute the operations of RFMs (refactorings) first on each Jak-like feature module individually and execute the refactored Jak-like feature modules afterwards ($(R(F_2) \bullet R(F_1))$, right perimeter in Fig. 4.4).[7]

If both approaches always yield equal programs, we can also reuse refactoring approaches, which work for stand-alone programs [CJ08, CN00, Fow99, FKK07, JH06, KK04, KKKS08b, MEDJ05], for SPLs. Further, if both approaches always yield equal programs, we can refactor all SPL products in one step by refactoring every Jak-like feature module individually. In the following, we prove whether both approaches indeed always yield equal programs.

We found examples, which indicate that approach (a) does not always yield equal programs to approach (b); we describe these examples in Section 4.2.1. To provide a convincing and general answer on whether approach (a) yields equal programs to approach (b) for all refactorings and to estimate the number of cases for which approach (a) yields equal programs to approach (b), we define a formal model of refactorings in Section 4.2.2 and prove algebraic properties of refactorings for the general case. We will find out that refactorings do not distribute over the operations of Jak-like feature modules in general; that is, both composer-tool implementation approaches (cf. Fig. 4.4) do not always yield equal programs. Based on our results, we conclude that the best approach for us to implement a composer tool is to execute Jak-like feature modules and RFMs consecutively in feature order (corresponds to approach (a); left perimeter in Fig. 4.4). During this analysis, we additionally prove properties that are important for later discussions of this thesis (e.g., we prove that there are identity operations in the domain of refactorings and that there are operations which invert each other in the domain of refactorings).

4.2.1. Cases of Nondistributivity of Refactorings

We now describe four examples in which executing Jak-like feature modules and RFMs consecutively (left perimeter in Fig. 4.4) does not yield the same results as

[7]This approach was inspired by composer tools of Jak-like feature modules [Bat06].

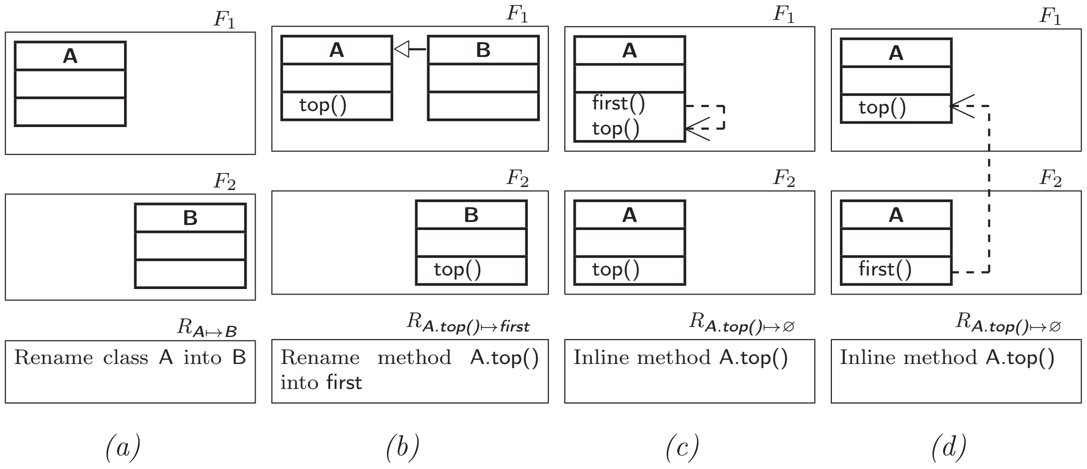


Figure 4.5.: *Cases in which refactorings do not distribute over feature modules.*

executing RFM operations first on the Jak-like feature modules (right perimeter in Fig. 4.4).

Consecutive execution of Jak-like feature modules and RFMs can result in a different failure than executing the RFMs' operations (i.e., the refactorings) first on the Jak-like feature modules. For illustration, suppose that a Rename-Class RFM $R_{\mathsf{A} \mapsto \mathsf{B}}$ follows two Jak-like feature modules F_2 and F_1 (cf. Fig. 4.5a). F_1 creates a class A; F_2 creates a class B; $R_{\mathsf{A} \mapsto \mathsf{B}}$ renames A into B. According to the preconditions of $R_{\mathsf{A} \mapsto \mathsf{B}}$, a class A must exist and no class B must exist in the input program of $R_{\mathsf{A} \mapsto \mathsf{B}}$.[8] Executing $R_{\mathsf{A} \mapsto \mathsf{B}}$ consecutively after F_2 and F_1 (i.e., $R_{\mathsf{A} \mapsto \mathsf{B}}((F_2 \bullet F_1))$) fails because class B exists in the generated program of $(F_2 \bullet F_1)$, which is the input program of $R_{\mathsf{A} \mapsto \mathsf{B}}$. Executing $R_{\mathsf{A} \mapsto \mathsf{B}}$ first on F_2 and F_1 individually (i.e., $(R_{\mathsf{A} \mapsto \mathsf{B}}(F_2) \bullet R_{\mathsf{A} \mapsto \mathsf{B}}(F_1))$) fails, too; while $R_{\mathsf{A} \mapsto \mathsf{B}}(F_1)$ succeeds, $R_{\mathsf{A} \mapsto \mathsf{B}}(F_2)$ fails because there is no class A in F_2. If we would tolerate $R_{\mathsf{A} \mapsto \mathsf{B}}(F_2)$ to fail, then class B of $R_{\mathsf{A} \mapsto \mathsf{B}}(F_2)$ would replace class B of $R_{\mathsf{A} \mapsto \mathsf{B}}(F_1)$, incorrectly. Summarizing, both implementation approaches fail but with different errors.

Consecutive execution of Jak-like feature modules and RFMs can succeed while executing the RFMs' operations first on the Jak-like feature modules fails. For illustration, suppose that a Rename-Method RFM $R_{\mathsf{A.top()} \mapsto \mathsf{first}}$ follows F_2 and F_1 (cf. Fig. 4.5b). F_1 creates a class A with a method A.top() and a class B with A as its superclass; F_2 refines B and creates method B.top(); $R_{\mathsf{A.top()} \mapsto \mathsf{first}}$ renames A.top() into first. Executing $R_{\mathsf{A.top()} \mapsto \mathsf{first}}$ consecutively after F_2 and F_1 (i.e., $R_{\mathsf{A.top()} \mapsto \mathsf{first}}((F_2 \bullet F_1))$) generates classes A and B with the methods A.first() and B.first().[9] Executing $R_{\mathsf{A.top()} \mapsto \mathsf{first}}$ first on F_2 and F_1 individually (i.e.,

[8] If there is no class A in the input program of $R_{\mathsf{A} \mapsto \mathsf{B}}$ then $R_{\mathsf{A} \mapsto \mathsf{B}}$ fails because there is no class to rename; if there is a class B then $R_{\mathsf{A} \mapsto \mathsf{B}}$ fails, too, because $R_{\mathsf{A} \mapsto \mathsf{B}}$ would create a second B and this is an error in most languages (cf. Sec. 2.3, p. 16).

[9] $R_{\mathsf{A.top} \mapsto \mathsf{first}}((F_2 \bullet F_1))$ renames B.top() because B.top() overrides renamed A.top().

$(R_{\mathsf{A.top()}\mapsto\mathsf{first}}(F_2) \bullet R_{\mathsf{A.top()}\mapsto\mathsf{first}}(F_1)))$ fails because A.top() does not exist in F_2. If we would tolerate $R_{\mathsf{A.top()}\mapsto\mathsf{first}}(F_2)$ to fail, then classes A and B would exist but with the methods A.first() and B.top(). Summarizing, one implementation approaches succeeds while the other one fails.

Consecutive execution of Jak-like feature modules and RFMs can succeed while executing the RFM's operation first on the Jak-like feature modules fails. For illustration, suppose that an Inline-Method RFM $R_{\mathsf{A.top()}\mapsto\varnothing}$ follows F_2 and F_1 (cf. Fig. 4.5c). F_1 creates method A.top() and method A.first() which calls A.top(); F_2 refines A.top(); $R_{\mathsf{A.top()}\mapsto\varnothing}$ inlines method A.top(). Executing $R_{\mathsf{A.top()}\mapsto\varnothing}$ consecutively after F_2 and F_1 (i.e., $R_{\mathsf{A.top}\mapsto\varnothing}((F_2 \bullet F_1)))$ generates A.first() which encapsulates the refined body of A.top(). Executing $R_{\mathsf{A.top()}\mapsto\varnothing}$ first on F_2 and F_1 individually (i.e., $(R_{\mathsf{A.top()}\mapsto\varnothing}(F_2) \bullet R_{\mathsf{A.top()}\mapsto\varnothing}(F_1)))$ fails; while $R_{\mathsf{A.top()}\mapsto\varnothing}(F_1)$ succeeds, $R_{\mathsf{A.top()}\mapsto\varnothing}(F_2)$ fails because the refinement of method A.top() cannot be inlined in F_2. If we would tolerate $R_{\mathsf{A.first()}\mapsto\varnothing}(F_2)$ to fail, then A.first() is generated which encapsulates the *unrefined* body of A.top(); further, A.top() of F_2 would refine a nonexisting method. Summarizing, one implementation approach succeeds while the other fails.

Consecutive execution of Jak-like feature modules and RFMs can succeed while executing the RFM's operation first on the Jak-like feature modules generates an erroneous program. For illustration, suppose that an Inline-Method RFM $R_{\mathsf{A.top()}\mapsto\varnothing}$ follows F_2 and F_1 (cf. Fig. 4.5d). F_1 creates method A.top(); F_2 creates method A.first() which calls A.top(); $R_{\mathsf{A.top()}\mapsto\varnothing}$ inlines A.top(). Executing $R_{\mathsf{A.top()}\mapsto\varnothing}$ consecutively after F_2 and F_1 (i.e., $R_{\mathsf{A.top}\mapsto\varnothing}((F_2 \bullet F_1)))$ generates A.first() which encapsulates the body of A.top(). Executing $R_{\mathsf{A.top()}\mapsto\varnothing}$ first on F_2 and F_1 individually (i.e., $(R_{\mathsf{A.top()}\mapsto\varnothing}(F_2) \bullet R_{\mathsf{A.top()}\mapsto\varnothing}(F_1)))$ fails; while $R_{\mathsf{A.top()}\mapsto\varnothing}(F_1)$ succeeds, $R_{\mathsf{A.top()}\mapsto\varnothing}(F_2)$ fails because there is no method A.top() to inline in F_2. Note that if we tolerate $R_{\mathsf{A.top()}\mapsto\varnothing}(F_2)$ to fail, an erroneous program is generated in which A.first() calls a nonexisting method. Summarizing, one implementation approach succeeds while the other generates an erroneous program.

4.2.2. Formal Proof of Algebraic Properties of Refactorings

The examples above show that executing Jak-like feature modules and RFMs consecutively yields different results in *some* cases than executing the RFM operations first on Jak-like feature modules before the refactored Jak-like feature modules are composed. To gain confidence in our implementation decision of the RFM composer tool, we must know whether the above examples are just *rare exceptions*. For that, we formally proved algebraic properties of refactorings in the setting of SPLs. For example, the algebraic property we prove in order to verify the assumption of Figure 4.4, page 36, is *distributivity* of refactorings over Jak-like-feature-module execution.[10]

[10] To prove that an algebraic property does not hold in general, a counterexample suffices. However, as we need to know the probability of these cases, a counterexample is not enough.

An Algebra for Feature Modules and Refactorings

To analyze refactorings algebraically with respect to feature modules, we define terms that describe programs, and operations that describe program transformations of Jak-like feature modules and RFMs.

Terms

Code ($\mathbb{Q}$). We represent code in our algebra because we want to compare whether programs are equal or not. We analyzed before (cf. Chap. 3) that scoped names of modules – but not method bodies – decide on module reuse when the functionality of that module does not change. For that, as a first step, we focus on refactorings which alter scoped names and we represent code in our algebra by a set of scoped names. For example, we describe the code of the Jak-like feature module *Directed* represented in Figure 4.2a, page 32, by the set of its scoped names: $Q_{Directed} = \{\mathsf{Vertex}, \mathsf{Vertex.display()::void}\}$. Set semantics suffices because the order of named pieces of code generally does not matter.[11] We use the meta-variable Q to represent a set of scoped names (i.e., to range over $\mathbb{Q}$).

Error state ($\mathbb{E} = \{\epsilon, \checkmark\}$). Programs, which include errors or were generated erroneously, do not implement the features that were selected by users. Thus, those programs should not be products of an SPL [CP06, KAT+09, TBKC07]. We annotate code with an error state in order to indicate in our algebra whether the annotated code was generated without error. The symbol ϵ indicates an error, whereas the symbol $\checkmark$ indicates that there was no error. We use the meta-variable e to represent an error state (i.e., to range over $\mathbb{E}$).

Programs ($\mathbb{F} = \mathcal{P}(\mathbb{Q}) \times \mathbb{E}$). In our algebra, a program is code, which was generated from feature modules and which is annotated with an error state. In our algebra, a Jak-like feature module is a program as well; but one that has not been generated before. As a result, the error state of a Jak-like feature module is always $\checkmark$.

If a program was generated by synthesizing feature modules and one or more feature-module executions failed, we annotate the program's code with ϵ. If a program was synthesized without error, we annotate its code with $\checkmark$. For example, the term for the program represented in Figure 2.5b, page 14, which was generated without error, is $\langle\{\mathsf{Vertex}, \mathsf{Vertex.name::String}, \mathsf{Vertex.predecessor::String}, \mathsf{Vertex.dweight::int}, \mathsf{Vertex.display()::void}\}; \checkmark\rangle$.

In our algebra, two programs are equal only when their code as well as their error states are equal. As a result, programs with equal code can be unequal when their

[11] The order of named pieces of code does only matter for initialized, static member variables in Java [GJSB05].

error states differ[12]:

$$
\begin{array}{lrcl}
((Q_1 \neq Q_2) \rightarrow & (\langle Q_1; e_1\rangle & \neq & \langle Q_2; e_2\rangle)) \\
 & (\langle Q_1; \checkmark\rangle & \neq & \langle Q_2; \epsilon\rangle) \\
 & (\langle Q_1; \epsilon\rangle & = & \langle Q_1; \epsilon\rangle) \\
 & (\langle Q_1; \checkmark\rangle & = & \langle Q_1; \checkmark\rangle)
\end{array}
\tag{4.1}
$$

We use the meta-variable F to represent a program (i.e., to range over $\mathbb{F}$).

Operations

Code composition ($\cup : \mathcal{P}(\mathbb{Q}) \times \mathcal{P}(\mathbb{Q}) \rightarrow \mathcal{P}(\mathbb{Q})$)**.** Jak-like feature modules compose code of an input program with the code they encapsulate. Feature modules add classes and class members to the input program and refine classes and class members of that input program. We described code by sets of scoped names and so the composition of code corresponds in our algebra to the union of simple sets. As an example, reconsider Figure 2.5a, page 14; the composition of the code of *Directed* ($Q_{Directed}$ = {Vertex, Vertex.name::String, Vertex.display()::void}) and the code of *ShortestPath* ($Q_{ShortestPath}$ = {Vertex, Vertex.predecessor::String, Vertex.dweight::int, Vertex.display()::void}) corresponds in our algebra to ($Q_{Directed} \cup Q_{ShortestPath}$) = {Vertex, Vertex.name::String, Vertex.predecessor::String, Vertex.dweight::int, Vertex.-display()::void}; this set exactly represents the code of the program generated from *Directed* and *ShortestPath* shown in Figure 2.5b. Note that this definition of code composition is not limited to FOP.

We focus on algebraic properties of refactorings that are applied to synthesized programs but do not focus on the synthesis of programs itself (this has been done elsewhere [ALMK10]). Thus, in the following, we assume code composition to succeed always (e.g., we assume that a class exists when it should be refined).

Program extension ($\oplus : F \times Q \rightarrow F$)**.** A refactoring generally replaces pieces of code in a program by other pieces of code – we thus describe such replacement operation later as a sequence of a code-removal operation and a code-generation operation. We call the code-generation operation of a refactoring *program extension*; for example, a refactoring that renames class Vertex into ADT, generates a class ADT.

Program extension succeeds to generate code when the scoped name of the code to generate does not exist (the scoped name is not occupied) in the input program and when the input program is free of error; program extension fails to generate code otherwise. With respect to our representation of code (set of scoped names), we define program extension by means of *distinct* sets. As a result, program extension succeeds when the set of scoped names of the input program is distinct from the set of scoped names to generate and when the input program is free of error; the program-extension operation (and so the refactoring) fails otherwise:

[12]The reason is simple: If an error occurs, we change the error state of a program but not the code.

$$(\langle Q_1; e_1\rangle \oplus Q_2) \quad = \quad \begin{cases} \langle (Q_1 \cup Q_2); \checkmark\rangle, & (Q_1 \cap Q_2 = \varnothing) \wedge (e_1 = \checkmark) \\ \langle Q_1; \epsilon\rangle, & \mathit{otherwise} \end{cases} \tag{4.2}$$

Jak-like-feature-module execution ($\bullet : F \times F \to F$). Jak-like feature modules compose code of an input program with the code they encapsulate. Jak-like-feature-module execution succeeds when this code composition succeeds (we defined code composition to always succeed; cf. p. 40) and when the input program and the Jak-like feature module are free of error; Jak-like-feature-module execution fails otherwise:

$$(\langle Q_1; e_1\rangle \bullet \langle Q_2; e_2\rangle) \quad = \quad \begin{cases} \langle (Q_1 \cup Q_2); \checkmark\rangle, & (e_2 = e_1 = \checkmark) \\ \langle Q_1; \epsilon\rangle, & \mathit{otherwise} \end{cases} \tag{4.3}$$

Program contraction ($\ominus : F \times Q \to F$). We call the code-removal operation of a refactoring *program contraction*; for example, a refactoring that renames class Vertex into ADT, removes a class Vertex.

Program contraction succeeds to remove code, when code with the scoped name to remove exists in the code of the input program and when the input program is free of error; program contraction fails to remove code, otherwise. With respect to our representation of code (set of scoped names), we define program contraction by means of *supersets*. As a result, program contraction succeeds when the set of scoped names of the input program is a superset of the set of scoped names to remove and when the input program is free of error; the program-contraction operation fails otherwise:

$$(\langle Q_1; e_1\rangle \ominus Q_2) \quad = \quad \begin{cases} \langle (Q_1 \backslash Q_2); \checkmark\rangle, & (Q_1 \cap Q_2 = Q_2) \wedge (e_1 = \checkmark) \\ \langle Q_1; \epsilon\rangle, & \mathit{otherwise} \end{cases} \tag{4.4}$$

For simplicity, we define that program contraction can remove code without removing code nested in the removed code. As a result, in our algebra, a refactoring which removes a scoped name from a set of scoped names does not remove any other scoped name automatically (though it might represent a piece of code which is nested in the removed piece of code). For example, removing the scoped name Vertex from the program represented by the set of scoped names {Vertex, Vertex.display()::void} becomes {Vertex.display()::void} and not the empty set.

Algebraic Properties of Refactorings ($R : F \times Q \times Q \to F$)

A refactoring executed on a program corresponds to the following operations in our algebra: The refactoring removes the set of scoped names Q_1 from the set of scoped names of the program and then joins the resultant set with a set of new scoped names Q_2. For example, a refactoring "Rename class Vertex into ADT" executed on a program, removes the set of scoped names Q_1 = {Vertex} from the code (set of scoped names) of that program and then joins the resultant set of scoped names

Case #1 $((Q_1 \cap Q_2 = Q_2))$**:**

$$\begin{aligned}
&R_{Q_2 \mapsto Q_2}(\langle Q_1; \checkmark \rangle) \\
&= (((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_2) && (4.5) \\
&= ((\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_2) && (4.4) \\
&= \langle (((Q_1 \backslash Q_2) \cup Q_2); \checkmark \rangle && (4.2) \\
&= \langle Q_1; \checkmark \rangle \\
&\square
\end{aligned}$$

Case #2 $((Q_1 \cap Q_2 \neq Q_2))$**:**

$$\begin{aligned}
&R_{Q_2 \mapsto Q_2}(\langle Q_1; \checkmark \rangle) \\
&= (((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_2) && (4.5) \\
&= ((\langle Q_1; \epsilon \rangle \oplus Q_2) && (4.4) \\
&= \langle Q_1; \epsilon \rangle && (4.2) \\
&\lightning
\end{aligned}$$

Figure 4.6.: *Proof: There are refactorings that do not change a program [from KKAS11].*

with the set of scoped names $Q_2 = \{\mathsf{ADT}\}$. When a refactoring aims to remove the elements of a set of scoped names Q_1 from the code of the refactoring's input program, then all scoped names in Q_1 must exist in that input program; when the refactoring aims to generate code with a set of scoped names Q_2 in the input program then no name in Q_2 is allowed to exist in the input program. As a result, we can describe a refactoring as a composite operation of program contraction and program extension. We use the meta-variable R to range over refactorings and write $R_{Q_1 \mapsto Q_2}(F_1)$ to indicate that R replaces the code Q_1 by the code Q_2 in program F_1:

$$R_{Q_1 \mapsto Q_2}(F_1) = ((F_1 \ominus Q_1) \oplus Q_2) \tag{4.5}$$

We now give a set of simplifications with respect to our proofs; simplifications which we make here on purpose and which still allow us to conclude on refactorings to the extent we desire: We do not review erroneous input programs as a starting point for refactorings as such programs lead to erroneous generated programs, which are not desired [CP06, KAT+09, TBKC07]. We consider a proof to have failed when an algebraic property holds under preconditions that imply an erroneous generated program because erroneous generated programs are not desired. We consider refactorings in a simplified form such that they only affect the scoped names they pass as parameters to their refactoring types (i.e., such that refactorings become enumerative). For example, Rename-Method refactorings alter the scoped names of methods which override the method to rename although their scoped names were *not* given as parameters [Fow99] – here, we focus on Rename *Monomorphic* Method which only renames the single method given as a parameter.

The upcoming proofs are structured as follows: Before we prove that an algebraic property holds or does not hold, we motivate this property. After we proved that an algebraic property holds or does not hold, we derive requirements for future tools that refactor SPLs or products of an SPL.

Theorem: There are identity operations in the domain of refactorings. If a refactoring is an identity operation, it does not alter its input program because, therein, it removes exactly these scoped names which it regenerates afterwards. If there are identity operations in the domain of refactorings, the following formula should hold for at least one refactoring: $R_{Q_2 \mapsto Q_2}(F_1) = F_1$.

Motivation

- Refactorings that regenerate their input programs can be removed without effects on the generated SPL products. This promises to reduce the time a tool needs to generate an SPL product when this generation involves to execute sequences of refactorings.

We prove the existence of identity operations in the domain of refactorings in Figure 4.6. In Case 1, we applied the Equations 4.5, 4.4, and 4.2 and could prove that a refactoring R, which replaces Q_2 by Q_2 in its input program, regenerates its input program when the code to replace exists in the input program. Case 2 proves that such refactoring does not always generate its input program; a refactoring does not generate its input program when the code to replace does not exist in the input program. Summarizing, there are refactorings that are identity operations conditionally (i.e., if they succeed).

Consequences. It is possible to reduce the time a tool needs to execute a sequence of refactorings (e.g., refactorings might be sequenced to generate a product of an SPL) by removing refactorings which implement identity operations. We elaborate on how to remove refactorings from refactoring sequences to reduce the time a tool needs to execute the sequences in Section 6.2. We elaborate on how to guarantee the success of sequenced refactorings in the SPL domain in Chapter 5.

Theorem: There are inverse operations in the domain of refactorings. Refactorings replace code with code that is equivalent to the replaced one with respect to the provided functionality. Once a refactoring executed, we can replace the replacement with the unrefactored code, which again naturally is equivalent with respect to the provided functionality. Thus, this second replacement operation should be provable to be a refactoring too and, in any case, should generate the un-refactored program [Rob99]. That is, if there are inverse operations in the domain of refactorings, the following formula should hold for at least one refactoring: $R_{Q_2 \mapsto Q_1}(R_{Q_1 \mapsto Q_2}(F_1)) = F_1$.

Motivation

- Refactorings that follow each other but together regenerate their input program can be removed without effects on the generated SPL product. This promises to reduce the time a tool needs to generate an SPL product when this generation involves to execute sequences of refactorings.
- Inverse refactorings would allow programmers to synchronize SPL products with according SPLs when the generation of the products involves refactorings.

Case#1 $((Q_1 \cap Q_2 = Q_2), (Q_1 \cap Q_3 = \varnothing))$**:**

$$\begin{aligned}
& R_{Q_3 \mapsto Q_2}(R_{Q_2 \mapsto Q_3}(\langle Q_1; \checkmark \rangle)) & \\
&= R_{Q_3 \mapsto Q_2}(((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3)) & (4.5)\\
&= ((((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.5)\\
&= (((\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.4)\\
&= ((\langle ((Q_1 \backslash Q_2) \cup Q_3); \checkmark \rangle \ominus Q_3) \oplus Q_2) & (4.2)\\
&= (\langle (((Q_1 \backslash Q_2) \cup Q_3) \backslash Q_3); \checkmark \rangle \oplus Q_2) & (4.4)\\
&= (\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_2) & \\
&= \langle ((Q_1 \backslash Q_2) \cup Q_2); \checkmark \rangle & (4.2)\\
&= \langle Q_1; \checkmark \rangle &
\end{aligned}$$

$\square$

Case#2 $((Q_1 \cap Q_2 = Q_2), (Q_1 \cap Q_3 \neq \varnothing), ((Q_1 \backslash Q_2) \cap Q_3 \neq \varnothing))$**:**

$$\begin{aligned}
& R_{Q_3 \mapsto Q_2}(R_{Q_2 \mapsto Q_3}(\langle Q_1; \checkmark \rangle)) & \\
&= R_{Q_3 \mapsto Q_2}(((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3)) & (4.5)\\
&= ((((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.5)\\
&= (((\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.4)\\
&= ((\langle (Q_1 \backslash Q_2); \epsilon \rangle \ominus Q_3) \oplus Q_2) & (4.2)\\
&= (\langle (Q_1 \backslash Q_2); \epsilon \rangle \oplus Q_2) & (4.4)\\
&= \langle (Q_1 \backslash Q_2); \epsilon \rangle & (4.2)
\end{aligned}$$

$\lightning$

Case#3 $((Q_1 \cap Q_2 = Q_2), (Q_1 \cap Q_3 \neq \varnothing), ((Q_1 \backslash Q_2) \cap Q_3 = \varnothing))$**:**

$$\begin{aligned}
& R_{Q_3 \mapsto Q_2}(R_{Q_2 \mapsto Q_3}(\langle Q_1; \checkmark \rangle)) & \\
&= R_{Q_3 \mapsto Q_2}(((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3)) & (4.5)\\
&= ((((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.5)\\
&= (((\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.4)\\
&= ((\langle ((Q_1 \backslash Q_2) \cup Q_3); \checkmark \rangle \ominus Q_3) \oplus Q_2) & (4.2)\\
&= (\langle (((Q_1 \backslash Q_2) \cup Q_3) \backslash Q_3); \checkmark \rangle \oplus Q_2) & (4.4)\\
&= (\langle (Q_1 \backslash Q_2); \checkmark \rangle \oplus Q_2) & \\
&= \langle ((Q_1 \backslash Q_2) \cup Q_2); \checkmark \rangle & (4.2)\\
&= \langle Q_1; \checkmark \rangle &
\end{aligned}$$

$\square$

Case#4 $((Q_1 \cap Q_2 \neq Q_2))$**:**

$$\begin{aligned}
& R_{Q_3 \mapsto Q_2}(R_{Q_2 \mapsto Q_3}(\langle Q_1; \checkmark \rangle)) & \\
&= R_{Q_3 \mapsto Q_2}(((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3)) & (4.5)\\
&= ((((\langle Q_1; \checkmark \rangle \ominus Q_2) \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.5)\\
&= (((\langle Q_1; \epsilon \rangle \oplus Q_3) \ominus Q_3) \oplus Q_2) & (4.4)\\
&= ((\langle Q_1; \epsilon \rangle \ominus Q_3) \oplus Q_2) & (4.2)\\
&= (\langle Q_1; \epsilon \rangle \oplus Q_2) & (4.4)\\
&= \langle Q_1; \epsilon \rangle & (4.2)
\end{aligned}$$

$\lightning$

Figure 4.7.: *Proof: There are refactorings that invert other refactorings [from KKAS11].*

We prove the existence of inverse operations in the domain of refactorings in Figure 4.7. In Case 1, we applied the Equations 4.5, 4.4, and 4.2 to generate a program with two sequenced refactorings that is equal to the input program. However in Cases 2 and 4, the generated programs were generated with errors and so the generated program of the inverse refactoring is not equal to the input program, which has no error. The proof shows that a refactoring can only be inverted by refactorings if it succeeded. Summarizing, there are refactorings that are inverse operations conditionally (i.e., if they succeed).

Consequences. It is possible to reduce the time a tool needs to execute a sequence of refactorings (e.g., they might be sequenced to generate a product of an SPL) by removing sequences of refactorings which invert each other and together generate the input program. We elaborate on how to remove refactorings from refactoring sequences to reduce the time a tool needs to generate an SPL product in Section 6.2. Tools can synchronize SPL-product code and SPL code when the generation of the products involves refactorings. We elaborate on how to synchronize products and SPLs in the context of RFMs in Section 6.1. We elaborate on how to guarantee the success of sequenced refactorings in the SPL domain in Chapter 5.

Theorem: There are refactorings that distribute over Jak-like-feature-module execution. We discussed two approaches for how to generate refactored products of an SPL (cf. pp. 35f): (a) RFMs and Jak-like feature modules could be executed consecutively in feature order, or (b) the refactorings of RFMs could be executed first on the Jak-like feature modules individually and the refactored Jak-like feature modules could be executed afterwards. The property, which must hold for both approaches to indeed yield equal programs, is distributivity of refactoring over Jak-like-feature-module execution: $R((F_2 \bullet F_1)) = (R(F_2) \bullet R(F_1))$

Motivation

- Distributivity would allow us to implement RFMs with two approaches: To execute Jak-like feature modules and RFMs consecutively in feature order, or to refactor Jak-like feature modules first and execute the refactored Jak-like feature modules afterwards.
- Distributivity would allow us to fit RFMs into existing FOP algebras without changing these algebras, because these algebras reorder sequenced feature modules [ALMK10, BS07].
- Distributivity would allow us to refactor SPLs using existing tools that already allow programmers to refactor stand-alone programs.

We prove that refactorings can distribute over Jak-like-feature-module execution in Figure 4.8. In Case 1, we applied the Equations 4.5, 4.4, 4.2, and 4.3 to generate equal programs by executing refactorings and Jak-like feature modules consecutively

Case #1 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 = Q_3), (Q_1 \cap Q_4 = \varnothing), (Q_2 \cap Q_4 = \varnothing))$**:**

$$
\begin{array}{lll}
R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark \rangle) & & \\
= (((\langle (Q_1 \cup Q_2); \checkmark \rangle \ominus Q_3) \oplus Q_4) & & (4.5) \\
= ((\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark \rangle \oplus Q_4) & & (4.4) \\
= \langle (((Q_1 \cup Q_2) \backslash Q_3) \cup Q_4); \checkmark \rangle & & (4.2) \\
= \langle (((Q_1 \backslash Q_3) & \cup (Q_2 \backslash Q_3)) \cup Q_4); \checkmark \rangle & \\
= \langle (((Q_1 \backslash Q_3) \cup Q_4) & \cup ((Q_2 \backslash Q_3) \cup Q_4)); \checkmark \rangle & \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet \langle ((Q_2 \backslash Q_3) \cup Q_4); \checkmark \rangle) & (4.3) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet (\langle (Q_2 \backslash Q_3); \checkmark \rangle \oplus Q_4)) & (4.2) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet (((\langle Q_2; \checkmark \rangle \ominus Q_3) \oplus Q_4)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.5) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark \rangle \oplus Q_4) & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.2) \\
= (((\langle Q_1; \checkmark \rangle \ominus Q_3) \oplus Q_4) & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.4) \\
= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark \rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & & (4.5)
\end{array}
$$

$\square$

Case #2 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 \neq Q_3), (Q_1 \cap Q_4 = \varnothing), (Q_2 \cap Q_4 = \varnothing))$**:**

$$
\begin{array}{lll}
R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark \rangle) & & \\
= (((\langle (Q_1 \cup Q_2); \checkmark \rangle \ominus Q_3) \oplus Q_4) & & (4.5) \\
= ((\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark \rangle \oplus Q_4) & & (4.4) \\
= \langle (((Q_1 \cup Q_2) \backslash Q_3) \cup Q_4); \checkmark \rangle & & (4.2) \\
\neq \langle ((Q_1 \backslash Q_3) \cup Q_4); \epsilon \rangle & & (4.1) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet \langle Q_2; \epsilon \rangle) & (4.3) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet (\langle Q_2; \epsilon \rangle \oplus Q_4)) & (4.2) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet (((\langle Q_2; \checkmark \rangle \ominus Q_3) \oplus Q_4)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.5) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark \rangle \oplus Q_4) & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.2) \\
= (((\langle Q_1; \checkmark \rangle \ominus Q_3) \oplus Q_4) & \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.4) \\
= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark \rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & & (4.5)
\end{array}
$$

↯

Figure 4.8.: *Proof: Refactorings do not always distribute over feature module execution [from KKAS11].*

and by executing refactorings on every Jak-like feature module individually first. However, Case 2 proves that distributivity does not hold in general for any refactoring; in Case 2, the scoped names to transform exist in the code of one Jak-like feature module ($\langle Q_1; \checkmark \rangle$) but not in the code of a second one ($\langle Q_2; \checkmark \rangle$). As a result, the refactoring fails for Case 2 in $\langle Q_2; \checkmark \rangle$ and generates an erroneous program, finally. We show the remaining cases of this proof in Appendix A. We can now count the results of all cases (in Fig. 4.8 and Appendix A) to find out that *in theory* in the majority of cases, refactorings do not distribute over Jak-like-feature-module execution; that is, distributivity holds in three cases but it does not hold in seven, and these three cases are special (i.e., four preconditions in average) compared to the seven cases for which distributivity does not hold (i.e., three preconditions in average). We conclude that the examples we discussed in Section 4.2.1 were no exceptions but examples of a general rule; refactorings distribute over feature modules in only rare cases, theoretically.

Consequences. As a first consequence, we can implement an RFM composer tool following both approaches shown in Figure 4.4 *if* the scoped names to remove exist in *every* Jak-like feature module *and if* the scoped names to generate do not exist in *any* Jak-like feature module (if the refactoring is not the identity operation). However, we do not want to restrict RFMs to SPLs in which this constraint holds because we expect this would rule out a number of SPLs. As a result, we implemented the RFM composer tool to execute Jak-like feature modules and RFMs consecutively in feature order (left perimeter in Fig. 4.4).

As a second consequence, we cannot generally reorder RFMs to regroup Jak-like feature modules and RFMs in a sequence of program transformations because this would require to execute refactorings on the individual, reordered, Jak-like feature module (we showed this is not possible in general). Hence, RFMs do not fit FOP algebras as is that aim to regroup transformations in transformation sequences [ALMK10, BS07].

Finally, there are consequences of our proof regarding distributivity beyond RFMs: We cannot use refactoring techniques of stand-alone programs [CJ08, CN00, FKK07, Fow99, JH06, KK04, KKKS08b, MEDJ05] without adaptation to refactor Jak-like feature modules of SPLs. The algebra, proofs, and consequences, which we discussed, are described in a way that we argue is applicable to more SPL approaches than RFMs (e.g., [SD10]).

Solutions. We proved that refactorings do not distribute over Jak-like-feature-module execution, in general. Tools, however, can implement a *compromise* to tackle the tasks to some extend, for which distributivity was desired. Specifically, tools can execute parts of refactoring operations at the level of Jak-like feature modules; but these tools also must analyze the reasons of failures, which may occur for the refactorings at the level of Jak-like feature modules, in a global check ($R_\int$):

$$R((Q_1 \cup Q_2)) = R_\int((R'(Q_1) \bullet R'(Q_2))) \tag{4.6}$$

Within the global check, the refactoring tool must distinguish *noncritical* and *critical* failures of refactorings at the Jak-like-feature-module level (i.e., at the level at which individual Jak-like feature modules are refactored). When a refactoring engine detects a noncritical failure, this engine must evaluate that at least one refactoring at the Jak-like-feature-module level does not fail for this reason. When a refactoring engine detects a critical failure, the refactoring engine must abort the refactoring immediately. Noncritical and critical failures are specific to the executed refactoring; to demonstrate this, we give two examples:

- A refactoring of the type Rename Class can fail at Jak-like-feature-module level because (a) the class to rename does not exist and/or (b) the class to generate exists. Failure (a) is noncritical because other Jak-like feature modules, which are refactored as well, might encapsulate the class to rename. In the global check, however, a refactoring tool must ensure that the refactoring does not fail due to failure (a) in all Jak-like feature modules; if the refactoring fails due to failure (a) in all Jak-like feature modules, then no class to rename exists in the whole SPL code and then failure (a) becomes critical. Failure (b) is critical from beginning. If the refactoring fails in one Jak-like feature module due to failure (b) then the class, which exists in this Jak-like feature module, may capture the name of a class that was created by the refactoring in another Jak-like feature module. A sophisticated refactoring engine might tolerate failure (b) when the tool can verify that a capture does not occur for any legal SPL product.

- A refactoring of the type Rename Method can fail at Jak-like-feature-module level because (a) the method to rename does not exist and/or (b) the method to generate exists – this is equivalent to Rename Class as discussed before. In addition to Rename Class, Rename Method transforms code of which the scoped name alone does not tell whether the code must be transformed, such as methods that override the parameter method. To determine whether a method should be transformed at the Jak-like-feature-module level, a refactoring operation at this level thus must have access to all class relations (inheritance/forwarding) and all class-member relations (e.g., overriding) – this is only possible for SPLs that ensure a common structure in *all* products (i.e., every class and every class member exists in every product).

Discussion on generality. We applied a number of simplifications to our proofs. We did not consider initialized, static member variables. We did not consider method bodies. We assumed that removing a scoped name does not affect other scoped names – even not scoped names of nested pieces of code. We focused on refactoring types that accept as parameters all scoped names that they alter. We modeled *program extension* and *Jak-like-feature-module execution* with set union, which is commutative, although program transformations in existing SPL approaches rarely commute – this fact does not impair our results, because we did not use commutativity for these operations in our proofs. The mentioned simplifications limit the generality of the proof cases which showed that a property holds. The proof cases, which showed that a property does not hold, (e.g., in the proof regarding distributivity) are not limited in generality and show that *in general* RFMs do not have certain algebraic properties.[13]

4.2.3. Design of the RFM Composer Tool

We prototypically implemented an RFM composer tool and integrated RFMs with the FOP language Jak (Jak extends Java; cf. Sec. 2.2.2). We learned from the above analysis of algebraic properties of refactorings for the implementation of the composer tool. Hence, we implemented the tool to iterate the feature modules in feature order and to execute them consecutively. A dispatcher calls a transformation engine of the AHEAD tool suite to execute Jak-like feature modules; the dispatcher calls refactoring meta-programs (partly reused from others [EH07]) to execute RFMs.

The RFM composer tool expects a refactoring unit to be implemented as a class. Such class refers to a refactoring meta-program using its API-artifact reference (keyword implements in Java) and implements the API artifact's methods to pass the refactoring parameters.

We prototypically implemented the RFM composer tool as a black-box framework such that programmers can add refactoring-type meta-programs in two steps: (a) adding a meta-program which implements the refactoring type and (b) passing the meta-program name to the composer as a command-line parameter.

4.3. Case Studies

Our main motivation to develop RFMs was to configure the structure of modules to ease their reuse (cf. Chap. 3). In this section, we report on studies that we performed in order to evaluate RFMs in different use cases. First, we evaluate whether RFMs help to integrate modules with incompatible environments. Second, we evaluate whether RFMs help to configure *nonfunctional properties (NFPs)* of SPL products.

4.3.1. Integration of Modules

We used RFMs to integrate stand-alone modules as well as products of module SPLs with incompatible environments [KBA09]. RFMs allowed us in a number of these studies to reuse modules in incompatible environments. We used RFMs to integrate a *stand-alone* module with an incompatible environment; we used RFMs to integrate products of *module SPLs* with an incompatible environment; we used *seldom-used* refactorings in RFMs to integrate a product of a module SPL with an incompatible environment; we used RFMs to integrate a *large-scale* module with an incompatible environment; we used RFMs to replace a wrapper; we used RFMs to replace a hierarchy of wrappers. We summarize the study in Table 4.2.

[13]We showed problems that occur in our simplified representations of code and refactorings; problems that do not vanish when completing the representations in complexity.

Table 4.2.: *Data on programs used to evaluate RFMs [extended from KBA09].*

Program/Module	**#SLOC**$^{\alpha}$	**Refactorings**
Log4J	~12K	1x Move Class, 2x Rename Method
ZipMe	~3K	2x Move Class, 1x Rename Class
Raroscope	~250	2x Move Class, 2x Rename Class
TrueZip	~13K	2x Move Class, 2x Rename Class, 1x Rename Method
GPL	~1K	4x Move Class, 2x Rename Class, 2x Rename Method, 6x Encapsulate Field, 2x Extract API Module, 1x Encapsulate In Package
Workbench.texteditor	~16K	1x Rename Class, 2x Rename Field
ADT$^{\beta}$ module	59	1x Rename Class, 1x Rename Method
JDOM	~6.7K	6x Rename Method, 6x Move Class, 2x Extract (Empty) Common Superclass, 1x Rename Class

$^{\alpha}$lines of source code; $^{\beta}$abstract data type

Stand-alone module. The database engine SmallSQL[14] (~20K lines of source code) uses a proprietary logging engine. Log4J[15] is a standard logging engine. Log4J cannot replace the SmallSQL logging engine due to incompatibilities with SmallSQL. We applied RFMs to Log4J in order to integrate Log4J with SmallSQL and in order to replace the SmallSQL logging engine by Log4J. After our integration, SmallSQL was able to use Log4J (and probably future releases of Log4J) to perform standardized logging.

To integrate Log4J, we transformed it into an SPL with one feature module (i.e., we moved the Log4J code into a Jak-like feature module). We extended this SPL by RFMs. One RFM moved class org.apache.log4j.Logger into the SmallSQL package smallsql.database; two RFMs renamed Logger methods to have SmallSQL-compatible names. The RFMs were simple to implement (i.e., we defined two parameters for each RFM) although the code they transformed was scattered throughout Log4J. For example, the Move-Class RFM automatically moved org.apache.log4j.Logger, automatically updated 144 references in 38 Log4J classes distributed over 10 Log4J packages, and automatically altered the access modifiers of numerous class members to be public.

We could not remove the following incompatibility with RFMs: We could not refactor Log4J to provide a single parameterless constructor in class org.apache.log4j.-Logger which calls set-access methods. We defined a Jak-like feature module to generate this constructor in Logger.

Module SPLs. We developed a user interface (153 lines of source code) which used an archive-access module to analyze files inside ZIP archives (file names, last modification dates, uncompressed file size) and to decompress them. We wanted to replace the archive-access module by the archive-access modules ZipMe, Raroscope, and

[14]`http://www.smallsql.de/` (accessed: July 10,2009)
[15]`http://logging.apache.org/log4j/` (accessed: July 10,2009)

TrueZip. ZipMe[16] allows users to access and analyze files in ZIP archives. Raroscope[17] allows users to access and analyze files in RAR archives. TrueZip[18] allows users to access and analyze files in TAR archives. The used versions of ZipMe and Raroscope were decomposed into Jak-like feature modules of an SPL and thus were configurable with respect to functionality (e.g., we could include or omit the code of feature *Checksum* from the archive-access modules). We moved the code of TrueZip into a Jak-like feature module to create an SPL with one feature module. All products of the ZipMe SPL, Raroscope SPL, and TrueZip SPL were incompatible with our user interface so we could not reuse them as is. After our integration, our user interface was able to analyze and decompress ZIP archives (using the old module and different ZipMe products), analyze RAR archives, and analyze and decompress TAR archives.

To integrate the archive-access-module products, we added numbers of RFMs. For example, we defined an RFM in the ZipMe SPL to rename class net.sf.zipme.ZipArchive into ZipFile. We could not remove the following incompatibilities with RFMs: We could not refactor ZipMe products and TrueZip products to make their archive-representation classes provide constructors that accept a single parameter object of type File; we defined a Jak-like feature module to generate such constructor in the ZipMe and TrueZip products. We could not refactor TrueZip to alter the signature of a method (i.e., to remove a parameter and assign it with a fixed value); we extended the Jak-like feature module which we already added to the TrueZip SPL to make it generate a suitable method (as a forward method). We could not refactor Raroscope products to provide the functionality to decompress RAR archives; we thus disabled archive decompression in our user interface for Raroscope (still we can analyze RAR archives).

Technically, the ZipMe SPL comprised 13 Jak-like feature modules and had 64 products; the Raroscope SPL comprised five Jak-like feature modules and had 16 products. We generated different ZipMe and Raroscope products and all were compatible automatically with our user interface when we selected the features, which triggered the RFMs and – in the case of ZipMe – the added Jak-like feature module.[19] This was especially interesting because just the fully-fletched products existed before.

Seldom-used refactorings. The GPL generates modules that contain graph data structures and graph algorithms. We wanted to use GPL products in an environment which used structures of the module OpenJGraph[20] to represent and manip-

[16]`http://sourceforge.net/projects/zipme/` (accessed: July 10,2009)
[17]`http://code.google.com/p/raroscope/` (accessed: July 10,2009)
[18]`https://truezip.dev.java.net/` (accessed: July 10,2009)
[19]We observed informally that with the fully-fletched ZipMe product, our user interface was able to decompress ZIP archives ∼4% faster than with the old, replaced ZIP-archive-access module. We observed that the fully-fletched ZipMe module had a ∼28% smaller footprint than the old, replaced ZIP-archive-access module. Integrating ZipMe thus payed off.
[20]`http://sourceforge.net/projects/openjgraph/` (accessed: July 10,2009)

ulate graphs (i.e., we wanted to replace OpenJGraph by GPL products). All GPL products, however, were incompatible with the environment which used OpenJGraph before. After our integration, we were able to reuse products of the GPL as modules in the environment.

To integrate the GPL products, we added RFMs to the GPL. These RFMs renamed GPL classes into OpenJGraph names and moved GPL classes into the OpenJGraph package salvo.jesus.graph. Interestingly, we also needed RFMs to implement seldom-used refactorings (e.g., to encapsulate GPL products in a package[21], encapsulate fields with access methods[22], and extract API artifacts).

We could not remove the following incompatibilities with RFMs: We could not refactor GPL products to provide serialization methods (toString methods) for the classes Vertex, Graph, and Edge. We could not refactor GPL products to provide a parameterless constructor in class Vertex. We could not refactor GPL products to alter the type which a parameter variable of a method had (to be double instead of int). We added a Jak-like feature module to the GPL; a Jak-like feature module which generates the above methods.

Technically, the GPL comprised 15 Jak-like feature modules and had 55 products. We generated different GPL products and all were compatible automatically with the environment when we selected the features, which triggered RFMs and the added Jak-like feature module.

Large-scale module. The Eclipse[23] module 'workbench.texteditor' (∼16K lines of source code) was reported incompatible with an environment [DNMJ08]. We transformed 'workbench.texteditor' into an SPL with one feature module by moving the Eclipse module's code into a Jak-like feature module. We added one RFM to the 'workbench.texteditor' SPL which renames class Levenshtein into Levenstein and added two RFMs which rename the fields DefaultCellComputer.levenshtein::Levenshtein and OptimizedCellComputer.levenshtein::Levenshtein into levenstein. After we executed the Jak-like feature modules and RFMs, 'workbench.texteditor' (and probably its future releases) was compatible with the formerly incompatible environment.

Replacing wrappers. An SPL [from BSST93] generated modules that contained abstract data types and wrapper code. We reimplemented this SPL using Jak and thereby reimplemented the wrapper code with an object wrapper. The wrapper allows programmers to access a class Container using the name Deque and allows programmers to access a method Container.insert_front(Element)::Element using

[21]This refactoring type has different names: Move Definition [Li06], Abstraction Extraction [Läm02], or Extract Package (see: `http://www.refactoring.com/catalog/extractPackage.html`; accessed: November 30,2010).

[22]An Encapsulate-Field refactoring generates a get-access and a set-access method for a member variable and updates all references to the field to use the get-access or set-access method [Fow99].

[23]`http://www.eclipse.org/` (accessed: July 16,2011)

the name add_front in class Deque. We added two RFMs to the SPL which renamed Container into Deque and renamed Deque.insert_front(Element)::Element into add_front. Since Container now is accessible using the name Deque and method Container.insert_front(Element)::Element now is accessible using the name add_front with objects of class Deque, we were able to remove the obsolete wrapper. Technically, this SPL comprised five Jak-like feature modules and had seven products.

Replacing wrapper hierarchies. We aimed at reimplementing object-wrapper hierarchies [from BCLvdS10] to integrate the module JDOM[24], which provides access to artifacts of extensible markup languages, with an environment that used the module XOM[25]; this environment consists of a number of JUnit[26] tests. We transformed the JDOM module into an SPL with one feature module by moving the JDOM module's code into a Jak-like feature module. Due to problems (which we describe next), we decided to integrate JDOM with just three of the mentioned JUnit tests. After the execution of the Jak-like feature modules and RFMs, we were able to reuse JDOM with three XOM JUnit tests; tests that yielded the same results as for wrappers. Thus, we call this study a *partial* success.

In this study, RFMs failed to completely reimplement the wrapper hierarchies and we now analyze why they failed. The major reason for RFMs to fail was that the wrappers reused a module to implement functionality that was missing in JDOM, and this module in turn required the existence of several JDOM classes. As a result, these JDOM classes had to exist even after the integration and refactoring of the JDOM module. To be precise, the wrappers reused the Jaxen[27] module (specifically, method org.jaxen.jdom.XPathNamespace.getJDOMNamespace()::org.jdom.Namespace) to extend the functionality of the JDOM method nu.xom.Namespace.getNamespace()::org.jdom.Namespace; the Jaxen module in turn required the JDOM class org.jdom.Namespace to exist. So we could not move org.jdom.Namespace to package nu.xom as desired, without creating incompatibilities between our JDOM product and the Jaxen module. A second reason for RFMs to fail was that the wrappers wrapped standard classes of Java such as java.util.List (wrapped in nu.xom.Nodes), java.lang.Integer (wrapped in nu.xom.Attribute_Type), or java.util.List (wrapped in nu.xom.Elements). We could not transform these standard classes with RFMs to be compatible because they were nontransformable in a referenced library. For the above reasons, we reused a number of wrapper classes and classes they reference, and moved them all into a new Jak-like feature module of the JDOM SPL.

As we replaced some wrapper classes with refactorings and refinements, some remaining wrapper classes had to become polymorph with their wrappee classes – our solutions obfuscated the class structure in several cases: We transformed the wrapper class nu.xom.Namespace and its wrappee class org.jdom.Namespace to inherit from a *new* common superclass (superclass extracted by an RFM); after we extended

[24]`http://www.jdom.org/` (accessed: March 25,2011)
[25]`http://www.xom.nu/` (accessed: March 25,2011)
[26]`http://www.junit.org/` (accessed: March 25,2011)
[27]`http://jaxen.codehaus.org/` (accessed: March 25,2011)

the new superclass, however, *both* Namespace classes had to provide new methods (i.e., Jak-like feature modules and RFMs intermixed in the feature order). The methods, which we had to add to the wrapper classes, could forward calls to wrappee methods. Some methods, which we had to add to the wrappee classes, however, could not be implemented meaningfully – we added a number of empty methods, accordingly. Furthermore, the wrapper class overrode wrappee methods which were annotated with final[28] and thus produced compiler errors; as neither the refactorings we implemented nor the refinements of Jak can alter final annotations, we had to adapt the original JDOM code, manually. In a similar situation, we added an RFM to rename a wrappee method such that the wrapper method with the same name and parameter list could be added later (both differed in their return types).

A difficulty we faced, though not a reason for the RFM failure, occurred when wrapper methods and constructors performed tests *before* they called the wrappee constructors. We could not refine JDOM constructors with this wrapper code because a constructor of a Java class must first call the superclass constructor instead of the tests [GJSB05]. As a workaround, we followed the advice from the Jak documentation [Sof08]: We extracted the constructor bodies into methods and refined these extracted methods accordingly.

After we executed the RFMs and Jak-like feature modules, the JDOM product no longer raised compiler errors with three JUnit tests that involved or tested wrapper classes which we replaced. We refined the JDOM code to add missing functionality (reused from the wrappers) before we executed these tests; as a result, these tests produced the same results as with the wrapper classes before (50 test cases ran without errors and with two failures; both failures occurred equally in the wrapper tests).

Summary. We observed that RFMs can help to integrate modules with environments with which the modules were incompatible before. As a result, in a number of cases, we were able to reuse the modules in these environments. We observed that RFMs support module integration even if modules, which are generated from SPLs, have never been generated before. The number of products of an SPL no longer matters for all these products to be adapted.

We observed that RFMs do *not replace* Jak-like feature modules but complement them with respect to module integration. We observed that (sequences of) RFMs can be selected as features in feature models just as Jak-like feature modules could be selected as features before.

We observed that renaming and moving classes and class members was most important in our studies. However, especially in the GPL study and the JDOM study, we observed the need for other refactoring types, too (e.g., in the GPL study, RFMs had to implement Extract-API-Artifact refactorings). In our perspective, RFMs can encapsulate any transformation that alters the structure of a program but does not alter its functionality.

[28]Methods annotated with final cannot be overridden in Java [GJSB05].

4.3.2. Configuration of Nonfunctional Properties

Programs differ in properties beyond functionality and structure (e.g., properties of presentation, power consumption, maintainability, binary size, or performance) [AT96, CG94, CIBR00, KCH$^+$90, Mye88, SKAP10, SRK$^+$08]. These properties are called *nonfunctional properties (NFPs)* [SKR$^+$08, SRK$^+$08]. Refactorings are known to affect NFPs [Fow99] and we show that RFMs can also be used to *configure NFPs* of SPL products [SKAP10]. For example, we show that a refactoring of type Rename Method can be used to configure code with respect to the NFP Code Quality by reducing the number of nonmnemonic method names. We summarize how refactorings of RFMs can be used to configure SPL products with respect to NFPs in Table 4.3, page 60.

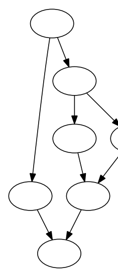
Code quality. RFMs can improve the quality of a piece of code when they improve this piece's accordance to desired naming conventions of different customers. For example, RFMs that implement the refactoring type Rename Method can remove the code smell of nonmnemonic method names [Fow99] or can allow a single module to satisfy different, conflicting naming conventions (e.g., of different customers).

RFMs can improve the quality of a piece of code when they improve this piece's accordance to code-quality metrics such as Cyclomatic Complexity. Cyclomatic Complexity evaluates the complexity of methods [McC76]; a method is considered to be complex when it includes a high number of branching statements such as conditionals or loops. In line with the metric authors [McC76], we found that executing Extract-Method refactorings can reduce the Cyclomatic-Complexity value of a method.[29] In addition, we found that refactorings of types Replace Conditional With Polymorphism [Fow99] and Replace Nested Conditional With Guard Clauses [Fow99] can reduce Cyclomatic Complexity because they can remove branching statements. We found that refactorings of types Inline Method and Replace Exception With Test [Fow99] may impair the Cyclomatic Complexity of a method, because these refactorings may introduce branching statements into methods. In Figure 4.9a, we show a method together with its control-flow graph; in Figure 4.9b, we show the same method with its control-flow graph after a Replace-Nested-Conditional-with-Guard-Clauses refactoring applied. According to the Cyclomatic-Complexity metric, the refactored version of the method (cf. Fig. 4.9b) is less complex.

RFMs can improve the quality of a piece of code when they improve this piece's accordance to code-quality metrics such as Weighted Methods Per Class. Weighted-Methods-Per-Class evaluates the complexity of a class [CK94]; a class is considered to be complex, when it encapsulates a number of methods. We found that refactorings of the type Extract Superclass can decrease the complexity of classes according to this metric (the refactorings separate methods of a class in new classes) and refactorings

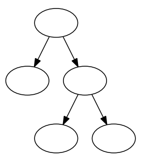
[29]To evaluate Cyclomatic Complexity of a method, we must analyze a control-flow graph of this method [McC76]. Specifically, we must subtract the number of vertices of the graph from the number of edges and sum to this number the doubled number of graphs of the method.

```
int setPredecessor(Vertex newP){
  int result;
  if (newP == null){
    result = −1;
  } else {
    if (newP == pred){
      result = −1;
    } else {
      pred = newP;
      result = 0;
    }
  }
  return result;
}
```



(a) Method and its control-flow graph before refactoring; Cyclomatic Complexity is 3.

```
int setPredecessor(Vertex newP){
  if (newP == null){
    return −1;
  }
  if (newP == pred){
    return −1;
  }
  pred = newP;
  return 0;
}
```



(b) Method of Fig. 4.9a and its control-flow graph after the refactoring; Cyclomatic Complexity is 1.

Figure 4.9.: *Code improvement according to NFP metric Cyclomatic Complexity achieved with a Replace-Nested-Conditional-with-Guard-Clauses refactoring.*

of type Collapse Hierarchy can impair the complexity of classes according to this metric (the refactorings join methods of different classes in one class).

Performance. RFMs can improve the performance of a piece of code (cf. Tab. 4.3). For example, an RFM which implements an Inline-Method refactoring can improve the performance of a program by removing method calls (and replacing them by the called method body); over-inlining, however, can impair the performance (e.g., by increased load times of huge methods) [DC94]. There might emerge additional refactoring opportunities after inlining a method [DC94] but we do not discuss them here. Conversely, an RFM which implements an Extract-Method refactoring can impair performance by introducing at least one method call toward the extracted method.

Footprint. RFMs can improve the footprint of a piece of code (cf. Tab. 4.3). For example, an RFM which implements a Pull-Up-Field refactoring can replace a number of member variables of subclasses by a single member variable of a common superclass; the refactored code thus has less footprint than the original. Conversely, an RFM which implements a Push-Down-Field refactoring multiplexes a member variable in subclasses and thereby might impair the footprint of the generated program.

Proof of Concept

In a proof-of-concept test, we executed an Inline-Method RFM to configure the NFP Performance of a micro benchmark program (developed by others [GP09]). However,

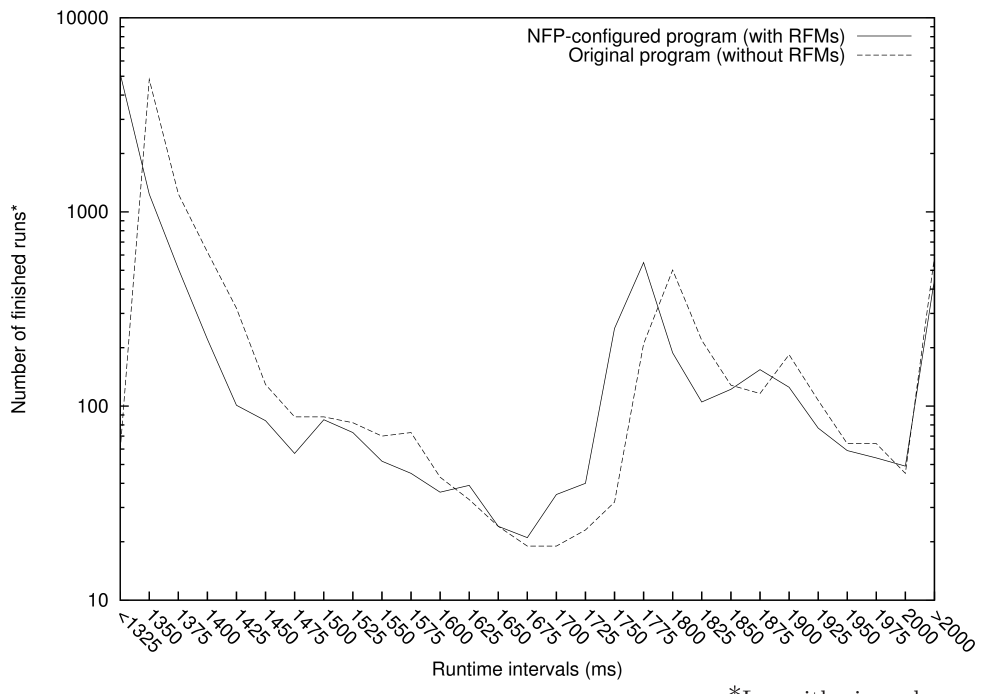


Figure 4.10.: *Runtimes of a program configured with RFMs for NFP Performance (with Inline-Method refactorings) [adapted from SKAP10].*

at first we did not gain any benefit. We conjectured that only the implementation and selection of high numbers of RFMs produce noticeable effects.

We argue that to implement and select high numbers of RFMs is laborious and error-prone; thus, we propose to adapt the product-generation process. We propose to configure an SPL product in two independent stages: In the first stage, users should configure the functionality and structure of their product; in the second stage, users should configure NFPs of their product. After the first stage, a preliminary program can be generated, hidden from the user, and analyzed automatically for pieces of code, which might be refactored to make the product better support certain NFPs. In the second stage, the refactorings and sequences of refactorings then can be grouped in the feature model according to the NFPs they alter and can be suggested to a user for the purpose of altering the NFP value. However, we argue to not generate RFMs that should improve the NFP Code Quality because this is controversial [Opd92, RBJ97].

We integrated the RFM composer tool with SPLConqueror[30], a tool which measures and optimizes NFPs for SPL products [SRK$^+$08]. We extended both tools to implement the above two-staged process. After that, we configured the same benchmark program as before (developed by others [GP09]) to expose the same NFP as

[30]`http://fosd.de/SPLConqueror/` (accessed: July 16,2011)

before (high performance). This time however, in the second stage of the configuration process, we selected the suggested surrogate feature *High Performance* and thus triggered 120 generated Inline-Method RFMs in one step. We re-evaluated the runtimes; we ran the original and the configured program 10,000 times and observed a performance benefit for the NFP-configured program.[31] We show our measurements in the diagram of Figure 4.10; the diagram relates runtime intervals to the number of runs that finished in an interval. The left-shift of the graph, which represents runtimes of the NFP-configured program, with respect to the graph, which represents the runtimes of the original program, indicates that the configured program required less time to run in general than the original program (the configured program was generally faster). For example, we observed that 59 of 10,000 runs of the original program finished in less than 1325ms; but, we observed that 5160 of 10,000 runs of the NFP-configured program finished in less than 1325ms. Summarizing, although the refactorings generated suboptimal code (e.g., they generated variable declarations not needed here), we gained an average runtime benefit of 2% through NFP configuration with RFMs. A continuative study conducted and supervised in the course of this thesis confirmed our results at a finer scale of granularity [Mos11].

We compared the footprint of the original benchmark program with the footprint of the NFP-configured counterpart. We observed that the sum of the footprints of all binary files (*.class* files in Java) of the NFP-configured program was higher by 18% than the sum of the footprints of all binary files of the original program (833,021 Byte compared to 705,569 Byte).[32] We thus showed that to configure with Inline-Method RFMs can also affect the footprint of a program.

4.4. Summary

In this chapter, we tackled the dilemma of module scalability which we described in Chapter 3: We extended techniques of SPL engineering to support an integrated, feature-driven configuration of a module's functionality and structure. This configuration allows module programmers to implement large-scale modules which are compatible with a lot of environments at the same time. Specifically, we described how refactorings can be used and implemented as selectable features of SPLs and called this approach *refactoring feature modules (RFMs)*. If a large-scale module is incompatible with an environment, users select features for this module; features which make the module compatible with the environment by altering the module structure. With the combination of Jak-like feature modules and RFMs, users can now configure the functionality of a module without worrying about the resulting structure (i.e., independently from the structure), because this structure can be configured and revised using RFMs. Users further can now configure the structure of a

[31] We performed the measurements of this section on an computer with an Intel Pentium 4 CPU with 3GHz and 2GB RAM running Windows XP SP2.

[32] Note that the increase in performance by 2% and the related increase in footprint by 18% can hardly be compared numerically – when a user requires a fast program, he/she might be required to accept footprint penalties.

module without worrying about the functionality of the module (i.e., independently from the functionality), because refactorings do not alter functionality. Summarizing, RFMs can aid the reuse of modules by removing incompatibilities.

We analyzed algebraic properties of refactorings with respect to Jak-like feature modules in order to decide, in which order a composer tool should execute Jak-like feature modules and RFMs best. We found that such tool should execute Jak-like feature modules and RFMs consecutively in feature order.

We evaluated RFMs with respect to module integration in a number of case studies. These studies showed that RFMs can help to integrate modules with incompatible environments. The studies however also demonstrated that RFMs do not replace but complement Jak-like feature modules. One study showed that we could not refactor a module when conflicting requirements exist regarding the structure of this module.

We demonstrated that RFMs can help to configure programs with respect to NFPs. Specifically, we demonstrated that with RFMs we are able to configure products of an SPL to provide high performance. We discussed that with RFMs we are able to configure products to have a small footprint or to adhere to different code-quality metrics. We observed that a single refactoring can improve one NFP of a program while at the same time it impairs another NFP of the same program. We observed that a single NFP can be improved by one refactoring and impaired by another refactoring.

Table 4.3.: *NFPs and how RFMs can alter them [extended and adapted from SKAP10].*

NFP	Improving Refactoring Types	Impairing Refactoring Types
Code quality		
Naming conventions	Rename Class, Rename Field, Rename Method	Rename Class, Rename Field, Rename Method
Method size	Extract Method, Create Template Method, Consolidate Duplicate Conditional Fragments	Inline Method
Cyclomatic complexity$^{\alpha}$	Extract Method, Replace Conditional With Polymorphism, Replace Nested Conditional With Guard Clauses	Inline Method, Replace Exception With Test
Weighted methods per class$^{\beta}$	Extract Class, Extract Subclass, Extract Superclass, Inline Method	Collapse Hierarchy, Consolidate Conditional expression, Decompose Conditional, Encapsulate Field, Extract Method, Hide Delegate, Inline Class
Depth of inheritance hierarchy$^{\beta}$	Replace Inheritance With Delegation, Collapse Hierarchy	Replace Delegation With Inheritance, Extract Subclass, Extract Superclass
Performance	Inline Method, Inline Class, Remove Middleman, Remove Setting Method, Replace Delegation With Inheritance, Replace Temp With Query, Inline Temp	Encapsulate Field, Extract Class, Extract Method, Form Template Method, Hide Delegate, Introduce Assertion, Decompose Conditional, Replace Inheritance With Delegation, Self Encapsulate Field, Change Unidirectional To Bidirectional
Footprint	Collapse Hierarchy, Pull Up Constructor Body, Pull Up Field, Pull Up Method, Remove Middleman, Remove Setting Method	Decompose Conditional, Encapsulate Field, Extract Class, Extract Interface, Hide Delegate, Inline Class, Inline Method, Inline Temp, Introduce Assertion, Introduce Explaining Variable, Push Down Field, Push Down Method, Remove Assignments to Parameters, Self Encapsulate Field

$^{\alpha}$introduced elsewhere [McC76]; $^{\beta}$introduced elsewhere [CK94]

5. Managing the Variability of Module Structure

Chapter 5 shares material with [KBK09].

Refactorings differ from refinements because refinements are monotonic (cf. Sec. 2.2.2) but refactorings are not, and refinements are enumerative (cf. Sec. 2.2.2) but refactorings are not. Refactorings are nonmonotonic because they add and remove code (e.g., if an RFM renames a class, this RFM in essence removes the old class and adds a new class, with equal functionality but a different name). Refactorings are nonenumerative because they do not always enumerate all the pieces of code that they alter (e.g., a Rename-Class RFM does not enumerate all methods of which it changes the scoped name or the return-type name). As a result, we must analyze whether programmers can still manage SPLs when these SPLs involve nonmonotonic, nonenumerative feature modules such as RFMs (i.e., whether programmers can still detect errors in their SPLs).

To analyze whether programmers can manage SPLs with RFMs, we developed and prototypically implemented a test to detect whether the programmer managed the variability correctly. If we can detect with this test at least one product that is in error, we know that the programmer has not managed variability correctly. Stated differently, SPL programmers must verify that every program, which they allow to generate, provides the features which the user selected for this SPL product. A product however only matches a set of features when every feature module succeeds that was executed on behalf of the features to generate the product. A feature module succeeds when its input program satisfies preconditions [TBKC07]. As a result, SPL programmers must validate all legal input programs of every feature module with respect to the preconditions of this feature module. But, a programmer cannot generate all legal input programs in advance as their number might be too high [Kru06]; especially, industrial SPLs with thousands of products [MPY$^+$04, vdLSR07] or even thousands of features [LP07, TSSPL09] are too large.

Safe composition is a technique to verify that every product of an SPL can be compiled and can be generated by the successful execution of feature modules [CP06, KA08, KAT$^+$09, KKB08, TBKC07]. Prior research on safe composition focused on Jak-like feature modules; thus, according approaches are insufficient for RFMs because they do not support nonmonotonic, nonenumerative feature modules.

In this chapter, we generalize existing concepts to verify safe composition for SPLs with nonmonotonic, nonenumerative feature modules such as RFMs. The RFMs we use to illustrate our concepts implement the refactoring types Rename Method and

Rename Class. The concepts however are not limited to Rename Method and Rename Class and presumably are not limited to RFMs in general. After we discussed the new concepts, we report on a tool which prototypically implements the concepts, and we report on a number of case studies.

The major insight of this chapter is that SPLs can benefit from nonmonotonic, nonenumerative feature modules, but are complex to manage by hand, when they include such modules. Interestingly, complexity grew rapidly in our studies, and more than we initially expected. We observed that concepts and prototypes as presented next are a must when nonmonotonic, nonenumerative feature modules are used in SPLs.

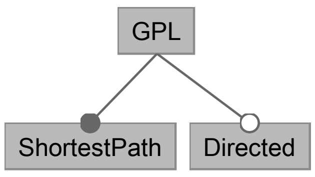
In the next steps, we review existing work on safe composition of Jak-like feature modules (cf. Sec. 5.1) and we analyze RFMs with respect to safe composition (cf. Sec. 5.2). We present our concept to test for safe composition of RFMs (cf. Sec. 5.3) and report on case studies (cf. Sec. 5.4). Finally, in Section 5.5, we discuss possible future extensions to the presented concepts.

5.1. Safe Composition of Jak-Like Feature Modules

A feature model describes products of an SPL by defining the meaningful combinations of features (cf. Sec. 2.2.1). In Figure 5.1a, we depict a simplified feature model of our GPL running example. The model describes that all legal GPL products must implement feature *ShortestPath* or must implement both features *ShortestPath* and *Directed*. A tool can map each of these features to a Jak-like feature module shown in Figure 5.2, and can execute each module when the SPL user selects the respective feature. We illustrate the feature order (i.e., the execution order for feature modules) in the diagram of Figure 5.1a by reading features from right to left (generally, this order can be defined separately from a feature diagram). According to the feature model depicted in Figure 5.1a, the Jak-like feature module *ShortestPath* can be executed alone to generate a legal SPL product or both Jak-like feature modules *ShortestPath* and *Directed* together can be executed to generate a legal SPL product; if both are executed, *Directed* executes before *ShortestPath* ((*ShortestPath* • *Directed*)) because this is defined in the feature order; *ShortestPath* cannot be executed before *Directed*, so (*Directed* • *ShortestPath*) does not yield a legal SPL product.[1]

The feature model of Figure 5.1a is inconsistent with the SPL implementation of Figure 5.2. According to the feature model, a program with only the feature *ShortestPath* is a legal product of the GPL. Such GPL product, however, cannot be generated with the feature modules of Figure 5.2 because feature module *ShortestPath* cannot be executed alone. Feature module *ShortestPath* depends on feature module *Directed* for two reasons: (a) *ShortestPath* refines class Vertex and, thus, depends on that Vertex exists in the input program; according to the feature modules of Figure 5.2 and the feature order, Vertex can only exist in the *ShortestPath* input

[1] To ease explanations in this chapter, we use the operator • to denote the sequencing of arbitrary feature modules of which each is denoted by its name.



(a) *Diagram notation.*

$$\begin{gathered} GPL \wedge \\ (ShortestPath \rightarrow GPL) \wedge \\ (GPL \rightarrow ShortestPath) \wedge \\ (Directed \rightarrow GPL) \end{gathered}$$

(b) *Formula notation.*

Figure 5.1.: *Feature model used to explain safe composition [adapted from KBK09].*

Feature module *Directed*

```
 1 public class Vertex {
 2    public String name;
 3    public void display(){
 4       ...
 5       System.out.println();
 6    } }
```

Feature module *ShortestPath*

```
 7 refines class Vertex {
 8    private String predecessor;
 9    private int dweight;
10    public void display() {
11       System.out.print( "Pred " + predecessor + " DWeight " + dweight + " " );
12       Super.display();
13    } }
```

Figure 5.2.: *Jak-like feature modules of the GPL [adapted from LHB01].*

program when *Directed* executed on this program; (b) *ShortestPath*-method Vertex.display()::void calls method Vertex.display()::void of its input program (Line 12) and, thus, depends on that Vertex.display()::void exists in the input program; according to the feature modules of Figure 5.2 and the feature order, the called method can only exist in the *ShortestPath* input program when *Directed* executed on this program. When *ShortestPath* executes without *Directed*, the required pieces of code Vertex and Vertex.display()::void do not exist and *ShortestPath* fails.

From the inconsistency between the feature model of Figure 5.1a and its associated feature modules of Figure 5.2, we know that we found an error: Either the feature model is in error, the feature module *ShortestPath* is in error, or both. An SPL programmer can now correct the reported inconsistency.

Feature diagrams as depicted in Figure 5.1a are used to illustrate feature models because they are easy to understand for SPL users. But, to validate consistency between a feature model and associated feature modules, the representation of feature models as propositional formulas is more helpful [MWC09]. In Figure 5.1b, we translate the feature model described in Figure 5.1a using standard translation rules [Bat05, CP06, CW07, TBKC07] into a propositional formula. In this formula, every variable corresponds to one feature – in line with preceding work, we thus call such variable *feature variable*. That is, the value of a feature variable is true when the feature the variable corresponds to is selected for an SPL product (i.e., when the according feature module is executed), and false if it is not selected. For example in Figure 5.1b, the value of feature variable *ShortestPath* is true when the feature *ShortestPath* is selected, and false otherwise. The propositional formula over feature variables evaluates to true when the assignment of feature variables in this formula corresponds to a feature combination that is legal to the SPL's feature model.

Prior research detects dependencies between Jak-like feature modules and formulates them in *execution constraints* [TBKC07]. If they detect an inconsistency between feature models and execution constraints of Jak-like feature modules they advise the domain engineer to add the violated constraint to the feature model. As a result, every feature combination that is legal to the revised feature model results in an SPL product that can be generated without error and can be compiled. To make the SPL feature model of Figure 5.1 consistent with the constraints of according feature modules of Figure 5.2, we can add the constraint that *ShortestPath* depends on *Directed*; prior research represents such constraint with a propositional formula over feature variables of according features[2]: $\mathit{ShortestPath} \rightarrow \mathit{Directed}$.

5.2. Analysis of RFMs for Safe Composition

Existing research does not verify safe composition for nonmonotonic, nonenumerative feature modules (i.e., feature modules which add code, remove code, and do not always enumerate all the pieces of code they transform). RFMs are nonmonotonic and nonenumerative, so we must generalize safe-composition concepts to cover RFMs. But, before we can generalize concepts of safe composition, we must analyze the properties of RFMs with respect to safe composition.

An RFM – in line with any other program transformation – executes safely when its input program fulfills preconditions (cf. Sec. 2.3, p. 16). The precondition of an RFM is commonly formulated in terms of scoped names which must exist or must not exist in input programs of that RFM. For example, a Rename-Class refactoring $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$, which renames class Vertex into ADT, imposes two preconditions on its input program: The input program must contain a class Vertex; the program must not contain a class ADT [Opd92, Rob99]. If there is no class Vertex, then $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$ fails as there is no class to rename. If there is a class ADT, then $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$ fails too as it cannot generate a second class ADT. If both preconditions of $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$ are fulfilled in every legal input program of $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$, then $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$ is verified to execute safely. Our aim is, thus, to formulate an execution constraint which guarantees that $R_{\mathsf{Vertex} \mapsto \mathsf{ADT}}$ executes safely and which we can verify with respect to a feature model.

In Figure 5.3, we show a Jak-like feature module *Directed* and five RFMs top-down in the order defined for their features in the feature model (i.e., in their execution order). We use these feature modules to demonstrate the challenges of formulating

[2]As we defined that there is only one order for all features, there is no need to encode this order in the propositional formula.

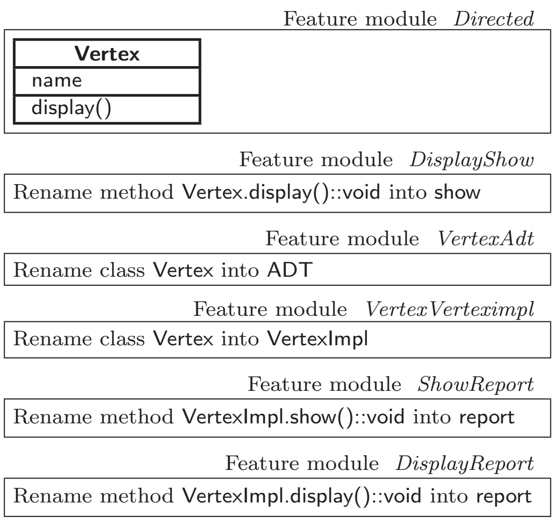

Feature module *Directed*
Vertex
name
display()
Feature module *DisplayShow*
Rename method Vertex.display()::void into show
Feature module *VertexAdt*
Rename class Vertex into ADT
Feature module *VertexVerteximpl*
Rename class Vertex into VertexImpl
Feature module *ShowReport*
Rename method VertexImpl.show()::void into report
Feature module *DisplayReport*
Rename method VertexImpl.display()::void into report

Figure 5.3.: *Jak-like feature modules and RFMs of a GPL version [adapted from KBK09].*

execution constraints for RFMs. Please note the rapidly growing complexity of execution constraints even for this small example:

- *Directed* transforms its input program (the empty program in this case) by adding a class Vertex. Vertex and its methods shall not reference other code so *Directed* has no preconditions. The execution constraint is: (*Directed*→true).

- *DisplayShow* transforms its input program by renaming method Vertex.display()::void into show. First, *DisplayShow* requires that a piece of code Vertex.display()::void exists in its input program. The piece exists when *Directed* executed before to generate the input program of *DisplayShow*; for that, *DisplayShow* can only be executed safely together with *Directed*. Second, *DisplayShow* requires a piece of code Vertex.show()::void to not exist. Vertex.show()::void cannot exist in the input program of *DisplayShow* because no feature-module sequence, which adheres to the feature order (i.e., to the feature-module execution order), can create show before *DisplayShow*; so, *DisplayShow* has no precondition in this respect. The composed execution constraint of *DisplayShow* is: (*DisplayShow*→*Directed*).

- *VertexAdt* transforms its input program by renaming class Vertex into ADT. First, *VertexAdt* requires that a piece of code Vertex exists in its input program. The piece exists when *Directed* executed before to generate the input program of *VertexAdt*; that is, *VertexAdt* can only be executed safely together with *Directed*. Second, *VertexAdt* requires a piece of code ADT to not exist. ADT cannot exist in the input program of *VertexAdt*; so, *VertexAdt* has no precondition in this respect. The composed execution constraint of *VertexAdt* is: $(\mathit{VertexAdt} \rightarrow \mathit{Directed})$.

- *VertexVerteximpl* transforms its input program by renaming class Vertex into VertexImpl. First, *VertexVerteximpl* requires that a piece of code Vertex exists in its input program. Second, *VertexVerteximpl* requires a piece of code VertexImpl to not exist. Interesting in this case: *VertexAdt* removes the required Vertex and thus can make *VertexVerteximpl* fail. For that, *VertexVerteximpl* can only be executed safely when *VertexAdt* is not executed. The composed execution constraint of *VertexVerteximpl* is: $(\mathit{VertexVerteximpl} \rightarrow (\neg \mathit{VertexAdt} \wedge \mathit{Directed}))$.

- *ShowReport* transforms its input program by renaming method VertexImpl.-show()::void into report. First, *ShowReport* requires that a piece of code VertexImpl.show()::void exists in its input program. Second, *ShowReport* requires a piece of code VertexImpl.report()::void to not exist. Interesting in this case: The required VertexImpl.show()::void exists only when both (a) the feature-module *sequence* $(\mathit{VertexVerteximpl} \bullet (\mathit{DisplayShow} \bullet \mathit{Directed}))$ executed to generate the input program *and* (b) *VertexAdt* did *not* execute to generate the input program. The composed execution constraint of *ShowReport* is: $(\mathit{ShowReport} \rightarrow (\mathit{VertexVerteximpl} \wedge \neg \mathit{VertexAdt} \wedge \mathit{DisplayShow} \wedge \mathit{Directed}))$.

- *DisplayReport* transforms its input program by renaming method VertexImpl.display()::void into report. First, *DisplayReport* requires that a piece of code VertexImpl.display()::void exists in its input program. Second, *DisplayReport* requires a piece of code VertexImpl.report()::void to not exist. Interesting in this case: The required VertexImpl.display()::void exists only when both (a) the sequence $(\mathit{VertexVerteximpl} \bullet \mathit{Directed})$ is executed and (b) *both* feature modules *DisplayShow* and *VertexAdt* are not executed. Here, the existence of VertexImpl.display()::void includes the nonexistence of VertexImpl.report()::void. The composed execution constraint of *DisplayReport* is: $(\mathit{DisplayReport} \rightarrow (\neg \mathit{ShowReport} \wedge \mathit{VertexVerteximpl} \wedge \neg \mathit{VertexAdt} \wedge \neg \mathit{DisplayShow} \wedge \mathit{Directed}))$.

Note that we can simplify above constraints when we remove transitive dependencies (i.e., dependencies toward feature modules which depend on each other). For example, we can simplify the constraint of *DisplayReport* to become $(\mathit{DisplayReport} \rightarrow (\mathit{VertexVerteximpl} \wedge \neg \mathit{DisplayShow}))$.

Summary. Safe composition for SPLs with Jak-like feature modules involves constraints in which one Jak-like feature module depends on only individual other Jak-like feature modules. It further involves constraints in which one feature module depends on only the execution of feature modules. Safe composition for SPLs with nonmonotonic, nonenumerative feature modules involves constraints in which one feature module can depend on the execution of a sequence of other feature modules. It further involves constraints in which one feature module can depend at the same time on a feature module X to be executed and on a feature module Y not to be executed. The reason for the increased complexity compared to Jak-like feature modules is that refinements of Jak-like feature modules are monotonic and enumerative but refactorings of RFMs are not.

5.3. Safe Composition of Nonmonotonic, Nonenumerative Modules

We now present new concepts for how a tool could verify safe composition for SPLs implemented with Jak-like feature modules and RFMs. First, in Section 5.3.1, we present the basic concept. After that, in Section 5.3.2 and Section 5.3.3, we discuss concepts that could be used to implement the basic concept. In our discussions, we use $\mathbb{F}$ to denote the set of all feature modules, $\mathbb{C}$ to denote the set of all sequences of feature modules over $\mathbb{F}$, $\mathbb{P}$ to denote the set of all propositional formulas over feature variables, and $\mathbb{S}$ to denote the set of all possible scoped names.

To keep explanations simple, we assume RFMs to be the final transformations in the feature order. This simplification reflects the use cases of RFMs discussed in Chapter 4 in which RFMs were proposed to be final transformations to integrate modules and to configure modules with respect to NFPs. As a first step, we concentrate on RFMs which implement refactoring types that depend only on scoped names and transform scoped names (especially important for module integration; cf. Sec. 4.3.1). We do not consider RFMs which implement refactoring types that always depend on method bodies (e.g., RFMs that implement the refactoring type Change Bidirectional Association to Unidirectional).[3] We cover these RFMs once we would extend our concepts to analyze method bodies (discussed in Section 5.5.1). Even with the above simplifications, our concepts can verify safe composition for implementations of 28% of all refactoring types from [Fow99].[4]

5.3.1. Basic Concept

An RFM implements a refactoring and executes safely when the refactoring executes safely (i.e., when the input program meets the preconditions of the refactoring); the RFM fails if the input program does not meet a precondition of the refactoring. A precondition may define that a scoped name must exist (i.e., the name is occupied

[3]Refactorings of type Change Bidirectional Association to Unidirectional remove member variables used for two-way relations between objects to create one-way relations [Fow99].

[4]We cover implementations of the refactoring types Add Parameter, Change Unidirectional Association to Bidirectional, Change Value to Reference, Encapsulate Field, Extract Class, Extract Complete Interface, Extract Superclass, Hide Delegate, Inline Method, Introduce Parameter Object, Move Class, Move Field, Move Method, Rename Class, Rename Field, Rename Method, Remove Assignments to Parameters, Replace Constructor with Factory Method, Replace Magic Number with Symbolic Constant, Replace Method with Method Object, Self Encapsulate Field. Two refactoring types, Consolidate Duplicate Conditional Fragments and Replace Nested Conditional with Guard Clauses, alter method bodies but not scoped names – as method bodies are irrelevant for module integration we count them to be supported.

by some code) or that a scoped name must not exist in an input program. In the following, we determine feature-module sequences which adhere to the feature order and make certain scoped names exist or not exist in the input programs of RFMs. We use those sequences to generate execution constraints. We transform the constraints into propositional formulas and use SAT solvers to validate the constraints in feature models for all legal feature combinations, in one step.

To simplify the next discussions, we define a function p to translate sequences of feature modules (denoted by their names) into propositional formulas ($p : \mathbb{C} \rightarrow \mathbb{P}$). The function p determines the feature of every feature module of its input sequence (for simplicity, we assume that the feature name is the feature-module name) and translates each of those features into a feature variable; p determines the features of all feature modules of the SPL, which are not part of the sequence, and translates each of those features into a negated feature variable; finally, p combines all translated feature variables and negated feature variables in a conjunction. For example, p translates the sequence of feature modules *VertexAdt* and *Directed* but not *DisplayShow* (i.e., ($VertexAdt \bullet Directed$)) into $VertexAdt \wedge \neg DisplayShow \wedge Directed$ (i.e., $p(VertexAdt \bullet Directed) = VertexAdt \wedge \neg DisplayShow \wedge Directed$). Features which follow *VertexAdt* in the feature order do not contribute to the formula because they are not relevant. The generated propositional formula evaluates to true for the feature-module sequence ($VertexAdt \bullet Directed$), and to false for others that adhere to the predefined feature order.

Refactorings have two types of preconditions: Some scoped names S^+ must exist and some scoped names S^- must not exist in input programs ($\mathrm{S}^+ \subseteq \mathbb{S}$; $\mathrm{S}^- \subseteq \mathbb{S}$; $\mathrm{S}^+ \cap \mathrm{S}^- = \varnothing$). For every scoped name which an RFM's refactoring requires to exist or not exist, we generate an execution constraint and validate it. For example, when an RFM R implements a refactoring which requires name $\mathsf{x} \in \mathrm{S}^+$ to exist in input programs, we validate that x exists in the input programs of R in the generation process of all legal SPL products; when R implements a refactoring which requires name $\mathsf{y} \in \mathrm{S}^-$ to not exist, we validate that y does not exist in the input programs of R in any legal SPL product. If all constraints are met by all input programs of an RFM in the generation process of all legal SPL products, this RFM is verified to execute safely.

In the following, we use *VertexVerteximpl* of Figure 5.3 as a running example for an RFM to verify. *VertexVerteximpl* renames class Vertex into VertexImpl. *VertexVerteximpl* executes safely, when the input program of *VertexVerteximpl* in the generation process of every legal SPL product includes code with scoped name Vertex and does not include code with scoped name VertexImpl.

Existence of scoped names. If a feature module R requires a scoped name x to exist, we assume a function c to calculate all feature-module sequences which (a) adhere to the feature order and (b) make code with name x exist in the input program of R ($c : \mathbb{S} \times \mathbb{F} \rightarrow \mathcal{P}(\mathbb{C})$). We discuss an implementation approach of c later.

$$c(\mathsf{x}, R) = \{C_1, C_2, \ldots, C_n\}$$

For example, *VertexVerteximpl* requires code with name Vertex to exist in the input program of *VertexVerteximpl.* With c, we can calculate all feature-module sequences which make Vertex exist:

$$c(\mathsf{Vertex}, \mathit{VertexVerteximpl}) = \{\mathit{Directed}, (\mathit{DisplayShow} \bullet \mathit{Directed})\}$$

We can now define execution constraints for R: If R is executed then one or more feature-module sequences in the result of $c(\mathsf{x}, R)$ must have been executed on the input program of R before. We determine the feature variable that corresponds to the feature of R and formulate the following constraint:

$$R \rightarrow (p(C_1) \vee p(C_2) \vee \ldots \vee p(C_n)) \tag{5.1}$$

We must validate these constraints for all legal SPL products to verify R to execute safely (i.e., for all legal feature combinations). For example, the constraint we must validate for *VertexVerteximpl* is[5]:

$$\mathit{VertexVerteximpl} \rightarrow (p(\mathit{Directed}) \vee p(\mathit{DisplayShow} \bullet \mathit{Directed}))$$

The constraints above are propositional formulas which a SAT solver can process efficiently [MMZ+01]. However, generally, SAT solvers only proof that at least one assignment exists for the variables of a formula such that the formula evaluates to true. To use SAT solvers, we transform Constraint 5.1 and the verification goal using a key insight from other researchers [CP06]: If C is a constraint that should hold in every legal feature combination and FM is a propositional formula which represents a feature model (cf. Fig. 5.1b, p. 63), then $FM \rightarrow C$ must be a tautology and $\neg(FM \rightarrow C)$ must be unsatisfiable. With respect to Constraint 5.1, we do no longer validate whether all legal SPL products fulfill Constraint 5.1 but we validate whether at least one legal SPL product exists (i.e., one legal feature combination), which does not fulfill Constraint 5.1. To verify Constraint 5.1, we thus finally validate that the following formula is unsatisfiable:

$$\neg(FM \rightarrow (R \rightarrow (p(C_1) \vee p(C_2) \vee \ldots \vee p(C_n)))) \tag{5.2}$$

If Formula 5.2 is unsatisfiable (i.e., the SAT solver fails), R executes safely. If Formula 5.2 is satisfiable (i.e., the SAT solver finds an assignment for the feature variables), R may fail when generating a legal SPL product. The assignment then tells us a legal feature combination that maps to a feature-module sequence for which R shall execute but for which the input program of R violates a precondition of R (because x does not exist).

To verify safe composition for *VertexVerteximpl*, we must validate that the follow-

[5] $p(\mathit{Directed}) = \neg\,\mathit{VertexAdt} \wedge \neg \mathit{DisplayShow} \wedge \mathit{Directed}$; $p((\mathit{DisplayShow} \bullet \mathit{Directed})) = \neg\,\mathit{VertexAdt} \wedge \mathit{DisplayShow} \wedge \mathit{Directed}$; feature modules executed after *VertexAdt* (predecessor of *VertexVerteximpl*) are not relevant because they cannot generate names in the input program of *VertexVerteximpl*.

ing formula is unsatisfiable:

$$\neg(FM \rightarrow (\mathit{VertexVerteximpl} \rightarrow (p(\mathit{Directed}) \vee p((\mathit{DisplayShow} \bullet \mathit{Directed})))))$$

Nonexistence of scoped names. If a feature module R requires a scoped name y to not exist, we use function c to calculate all feature-module sequences which adhere to the feature order and make code with name y exist in the input program of R.

$$c(\mathsf{y}, R) = \{C'_1, C'_2, \ldots, C'_n\}$$

For example, *VertexVerteximpl* requires code with name VertexImpl to not exist in the input program of *VertexVerteximpl*. With c, we can calculate that the set of feature-module sequences, which make VertexImpl exist in the input program of *VertexVerteximpl*, is empty ($c(\mathsf{VertexImpl}, \mathit{VertexVerteximpl}) = \varnothing$).

As before, we can define execution constraints for R: If R is executed then no feature-module sequence in the result of $c(\mathsf{y},R)$ is allowed to have been executed on the input program of R before:

$$R \rightarrow \neg(p(C'_1) \vee p(C'_2) \vee \ldots \vee p(C'_n)) \tag{5.3}$$

We must validate these constraints for all legal SPL products to verify R to execute safely. Again, we transform the constraint and the verification goal into a propositional formula and use a SAT solver to process both. As before, R executes safely if the formula $\neg(FM \rightarrow C)$ is unsatisfiable – this time with C being the Constraint 5.3. If the formula is satisfiable, the assignment for the feature variables tells us a legal feature combination for which R shall execute but for which the input program of R violates a precondition of R (because y exists).

For *VertexVerteximpl*, $c(\mathsf{VertexImpl}, \mathit{VertexVerteximpl})$ computes the empty set such that Constraint 5.3 becomes a tautology and the negated formula passed to the SAT solver is unsatisfiable.

If any verification above for the existence or nonexistence of scoped names fails (i.e., a SAT test succeeds), we know the feature model is in error, the feature modules are in error, or both. That is, a feature module R may fail although it is executed only to generate legal SPL products (in those products, the input program of R violates a precondition of R). The verification tool then can alert programmers and domain engineers to repair the SPL. We redo the above process until no constraint is violated any more (i.e., all SAT tests fail) – as a result, every legal feature combination generates an SPL product that can be compiled and can be generated without error.

5.3.2. Computing Input Programs that Encapsulate Scoped Names

In the last discussion, we assumed a function c exists to calculate feature-module sequences which adhere to the feature order and make a particular scoped name exist in the program they generate. A trivial approach to implement c would be to execute all sequences, which adhere to the feature order, and to test for the scoped

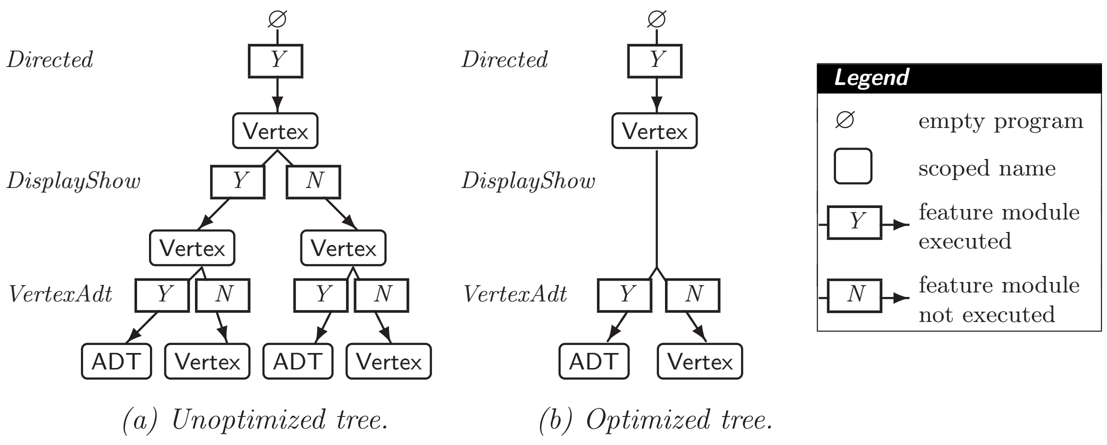


Figure 5.4.: *Tree to record the effect of RFM decisions on scoped name* Vertex *before feature module VertexVerteximpl of Fig. 5.3 executes [adapted from KBK09].*

name in their generated programs; however this approach does not scale because a single SPL can have millions of those sequences and we cannot execute them all. Thus, we investigated in concepts of a different approach to implement c that do not involve the execution of feature-module sequences.

We propose to create a decision tree for every scoped name that Jak-like feature modules can create (i.e., each scoped name is recorded in the root of a new decision tree). Thereby, we create multiple decision trees when different Jak-like feature modules create different pieces of code with equal scoped names. Once created, we record in each tree the effects of executing feature modules on scoped names, and of not executing them (corresponds to the selection of according features and of not selecting them). While the recording of a decision to execute a feature module may lead a tree node to a new node with a new scoped name, the recording of a decision to not execute a feature module always leads to a new node with the same scoped name. One decision tree thus represents one piece of code and records those scoped names which this piece has over time (after RFMs executed). Decision trees, finally, may include special nodes to indicate a scoped name got removed instead of transformed.

For example in Figure 5.4a, we depict how the scoped name Vertex is affected by decisions on feature modules which can execute prior to *VertexVerteximpl* according to the feature order (i.e., how decisions on *Directed*, *DisplayShow*, and *VertexAdt* affect Vertex; cf. Fig. 5.3). Arrows in this figure represent decisions on the execution of feature modules which lead from nodes of old scoped names (arrow source) to nodes of new scoped names (arrow target). Arrow annotations indicate whether the decision to execute a feature module generates the new scoped name or whether the decision to not execute a feature module does. If a feature module does not transform a certain scoped name the decision arrows, which start from this scoped name's node, lead to nodes with equal scoped names.

We propose to create decision trees by iteratively analyzing all feature modules in the order defined for their features in the feature model. First in each iteration, we verify using the existing decision trees that the current feature module executes safely (i.e., the algorithm knows all preconditions and transformation effects). Second in each iteration, we extend the existing decision trees and record in them how scoped names are altered when the current feature module is executed and when it is not executed. As a result, when we validate a feature module, the used decision trees include only effects of feature modules which can execute prior to the current feature module – this is on purpose because feature modules only transform code which other feature modules generated *before* (cf. Sec. 2.2.2). In each iteration step, we thus add one level of nodes to all decision trees.

With decision trees for all pieces of code which have a scoped name, we can calculate the feature-module sequences which adhere to the feature order and make a particular scoped name exist in the input program of a feature module to verify. When a feature module requires code with a particular scoped name to exist in its input program, we find all *leaf nodes* with that scoped name. We find the feature-module sequence, which adheres to the feature order and makes the scoped name of a leaf exist in the input program of the currently verified feature module, by analyzing the decisions along the path from this leaf backward to the empty program. Note that leafs with equal scoped names can exist in different trees, but these names can only exist in different products (i.e., the feature-module sequences recorded for them are always different) – otherwise the verification of a preceding feature module would have raised an error already because the nonexistence preconditions of this preceding module preclude the generation of equal scoped names in one product. That is, equal scoped names that are recorded in different trees are generated by preceding feature modules but do never occur in the same SPL product.

As an example, we now can use the decision trees to recalculate the set of feature-module sequences which adhere to the feature order and make Vertex exist in the input program of *VertexVerteximpl* of Figure 5.3 (in Sec. 5.3.1, we assumed a function c for this task to exist). Figure 5.4a shows the relevant decision tree with the recorded decisions on those feature modules which can execute before *VertexVerteximpl*. In that tree, two Vertex leaf nodes exist and thus we can calculate two different feature-module sequences which make Vertex exist in the input program of *VertexVerteximpl*, $c(\mathsf{Vertex}, \mathit{VertexVerteximpl}) = \{\mathit{Directed}, (\mathit{DisplayShow} \bullet \mathit{Directed})\}$. We translate and use these sequences to derive the execution constraint of *VertexVerteximpl*; finally, we verify this constraint with a SAT solver.

We encode the feature order in the decision trees because we iterate feature modules in the order defined for their features to record their effects. As a result, we do not have to encode this order again in the propositional formulas. A combination of features as defined in the constraints together with the predefined feature order (cf. Sec. 2.2.2) corresponds to a single feature-module sequence. For example, we created the decision tree of Figure 5.4a by iterating the feature modules *Directed*, *DisplayShow*, and *VertexAdt* in the order defined for their features. As a result, we know that both propositional formulas, $(\mathit{Directed} \wedge \mathit{DisplayShow})$ and $(\mathit{DisplayShow} \wedge \mathit{Directed})$, correspond to the single feature-module sequence

(*DisplayShow* • *Directed*) in a constraint to verify and a SAT-test result (i.e., we can use either one to generate constraints). In Section 5.5.2, we discuss how we could verify SPLs that have no feature order.

For understandability, we concentrate on the basic concepts only. That is, we omit discussions on special cases of single refactoring types which require extensions to the above concepts.[6]

Compression. Every decision tree is binary and balanced and we add a level of nodes to all trees each time we record the effects of a single feature module; as a result, the number of tree nodes grows exponentially in the number of recorded feature modules. Thus, memory and performance problems may arise when the number of feature modules is high or the number of decision trees is high. To avoid such problems, we compress decision trees: We only record the effects of decisions on the execution of a feature module for a leaf node when the executed feature module alters the scoped name of that leaf (i.e., when both decisions on the feature module do not lead the leaf to new nodes with equal scoped names). For example, the executed Rename-Method RFM *DisplayShow* of Figure 5.3 does not alter the scoped name of class Vertex; thus, both decisions on *DisplayShow* lead Vertex nodes to new nodes with equal scoped names in the decision tree of Figure 5.4a. Due to our compression, we do no longer record the effect of *DisplayShow* decisions for Vertex nodes; we depict the resulting compressed decision tree in Figure 5.4b. We observed in our studies (see Sec. 5.4) that this optimization is important because – especially in large-scale SPLs – most feature modules do not alter the scoped name of a particular piece of code.

With compressed decision trees we now calculate *patterns* for feature-module sequences when we calculate tree paths from leaf nodes toward the empty program; every pattern then only includes a decision on a feature module when this feature module decides whether the piece of code of the tree exists with the scoped name of the leaf. For example in Figure 5.3, feature module *DisplayShow* does not decide whether the piece of code Vertex (added by *Directed*) exists with the scoped name Vertex and so neither the tree of Figure 5.4b nor the pattern derived from this tree for scoped name Vertex includes a decision on *DisplayShow* (i.e., the path in the tree of Fig. 5.4b from the Vertex leaf toward the empty program does not include a decision on *DisplayShow*). We use the patterns instead of definite feature-module sequences to generate execution constraints.

Patterns translate differently into propositional formulas than definite feature-module sequences and so we must change our basic concept to use a new function *p'* instead of the function *p*.[7] In contrast to *p*, function *p'* translates single patterns

[6]For example, we must combine decisions of feature-module sequences of different leafs to verify and record the effect of an Add-Parameter refactoring. Specifically, we must combine decisions of sequences, which make the scoped name of the method to extend exist, with decisions of sequences, which make the scoped name of the class exist that shall be referenced in the new parameter declaration.

[7]Function *p* translates sequences of feature modules into propositional formulas (cf. Sec. 5.3.1).

into a propositional formula ($p' \colon \mathcal{P}(\mathbb{F}) \times \mathcal{P}(\mathbb{F}) \rightarrow \mathbb{P}$); that is, it translates a set feature modules decided to be executed and a set of feature modules decided to not be executed into a propositional formula. Thereby, p' determines the feature of each feature module decided in the pattern to be executed and translates this feature into a feature variable; p' determines the feature of each feature module decided to not be executed and translates this feature into a negated feature variable; finally, p' combines all translated feature variables and negated feature variables in a conjunction. If a pattern does not define a decision for a feature module, no feature variable for the feature of this feature module contributes to the translated formula; that is, the feature decision is left undefined (this is different from p which made the translated formula include a variable for every feature). For example, the pattern to describe the feature-module sequences which adhere to the feature order and make Vertex exist in the input program of *VertexVerteximpl* is ($\neg\, \mathit{VertexAdt} \wedge \mathit{Directed}$) (cf. Fig. 5.4b). The translated formula contains no feature variable for *DisplayShow* because *DisplayShow* does not decide the existence of the scoped name Vertex in the generated program. Again the translated propositional formula evaluates to true for every feature-module sequence which adheres to the feature order and makes Vertex exist in the input program of *VertexVerteximpl*, and false for others that adhere to the feature order.

To compress decision trees even more, we can merge their nodes when these nodes represent equal scoped names. As a result, decision trees become directed graphs when different feature-module sequences make one piece of code expose equal scoped names. As a second result, different decision graphs can share nodes, when different pieces of code can be transformed to expose equal scoped names. However, we show in Section 5.3.3 that we cannot merge nodes with equal scoped names in general.

We implemented additional optimizations; but we omit their discussion here for the clarity of this thesis – they are not important for the presented concept.[8]

5.3.3. Preconditions on Inheritance Hierarchies

A number of refactorings require more than the sole existence of certain scoped names in order to execute safely. To be precise, 14 of the 23 refactoring types, which we already cover for safe composition, require that in their input programs inheritance hierarchies fulfill preconditions. If the preconditions are not fulfilled, the refactorings fail though the refactored program may compile (cf. name capture, Sec. 2.3, p. 16). For example, a refactoring which renames the method Vertex.display()::void into show, requires that no method show must exist (a) in class Vertex but also (b) in any superclass or subclass of Vertex. If such superclass or subclass method exists, the refactoring may accidentally redirect method calls and thus change functionality. In order to verify preconditions on inheritance hierarchies, we must consider relations between pieces of code.

[8] For example, we remove the inner nodes of decision trees and instead attach precomputed parts of tree paths to leaf nodes.

We can detect the relations between pieces of code in Jak-like feature modules and maintain these relations for every node in the decision trees. For example, a node, which represents a method in one decision tree, references a node of a different tree, which represents the method's host class. Since a relation between tree nodes represents a relation between their respective pieces of code, we can now calculate those methods from node relations, which a method to be renamed may capture. As a result, we can combine patterns of related nodes to verify relations between according pieces of code: We can collect into a set $\mathbb{X}$ those nodes which have a certain scoped name, can calculate for each node in $\mathbb{X}$ those nodes it relates to, and can finally combine the patterns of related nodes to verify whether in a legal SPL product at least one node in $\mathbb{X}$ applies together with its related node before a certain feature module executes (i.e., both according, related pieces of code exist).

The above extension allows us to verify preconditions toward inheritance hierarchies correctly even if different pieces of code have equal scoped names (in different SPL products): Assume a method m_1, which has the scoped name m in an SPL product, is hosted by a class which has a superclass. Assume further a *different* method m_2, which has the scoped name m in a different SPL product, is hosted by a class which has no superclass/subclass. While m_1 may capture the name of a superclass method, m_2 cannot. As a result, the pattern which leads to the decision tree node of m_1 must be combined with patterns of methods which m_1 relates to and may capture; the method m_2 can be calculated to never relate to a method it may capture so the pattern of m_2 can be used as is.

As tree nodes now reference each other, two nodes are equal only (i.e., represent equivalent code with respect to refactoring success) when their scoped names are equal *and* their relations to other nodes are the same. As a result, we cannot merge nodes in general just because their scoped names are equal. We can merge nodes when their scoped names are equal *and* their relations to other nodes are the same (e.g., when two method nodes reference the same host-class node).

5.4. Case Studies

We extended existing algorithms and tools [TBKC07], which verify safe composition for Jak-like feature modules, in order to prototypically implement the safe-composition concepts presented in Section 5.3.[9] We evaluated our concepts and prototype in five case studies in order to show the feasibility of the presented concepts: To prove the concept, we studied a module SPL of small-scale abstract data types which was written independently of this work. To analyze whether storing high numbers of trees exceeds memory, we studied an SPL version (which uses RFMs) of the Eclipse module 'workbench.texteditor' with large-scale Jak-like feature modules.

[9]We reused the tool, which verifies Jak-like feature modules (developed by others [TBKC07]), and the tools and modules it depends on (e.g., the Sat4j SAT solver, bcj2j, and bccompiler). Accordingly, we had to adapt the studies slightly (e.g., we moved all classes into the default package first). The concepts, however, are not restricted this way.

Table 5.1.: *Data on SPLs used to evaluate our approach to safe composition [from KBK09].*

Program	**Refactorings**	**#SLOC**$^\alpha$	**#JFMs**$^\beta$	**#sn**$^\chi$ **from JFMs**$^\beta$	**#sn**$^\chi$ **from RFMs**	**max. TH**$^\delta$	**avg. TH**$^\delta$
ADT$^\epsilon$ module	1x Rename Class, 4x Rename Method	11	1	7	9	5	2
Workbench.texteditor	1x Rename Class, 2x Rename Field	~16K	2	3428	68	3	1.02
Workbench.texteditor #2	27x Rename Class, 28x Rename Field	~16K	2	3428	2538	55	1.53
GPL	2x Rename Class, 2x Rename Method, 18x Encapsulate Field, 2x Extract Interface$^\zeta$	~1K	15	160	252	4	1.89
ZipMe	1x Rename Class	~3K	14	656	18	2	1.03
Raroscope	2x Rename Class	~250	5	57	54	2	1.9

$^\alpha$lines of source code without RFMs; $^\beta$Jak-like feature module; $^\chi$scoped name; $^\delta$tree height; $^\epsilon$abstract data type; $^\zeta$API artifact

To check whether our algorithm terminates in a reasonable time on current computers for a high number of RFMs and a high number of trees, we studied an extended version of the large-scale 'workbench.texteditor' study. To test whether our concepts apply for an SPL in which a high number of Jak-like feature modules can execute *before* RFMs execute, we studied a version of the GPL. Finally, to analyze whether our algorithm terminates for SPLs with high numbers of SPL products, we studied safe composition for RFMs added to existing feature-oriented designs of archive-access modules. We summarize interesting properties of the studied SPLs in Table 5.1.

Proof of concept. We studied a version of a module SPL of abstract data types which leans on a study from prior work (cf. Sec. 4.3.1). With the study, we want to test now our safe-composition concept and prototype. The study has a feature model of moderate complexity (e.g., it subdivides features) and has a very small-scale code base (only 11 lines of source code; i.e., no functionality) – these properties made us believe that this study is simple. As a result, we were surprised that our verifier alerted errors because the study looked correct. After we analyzed the SPL in more detail, we found the error in the feature model and corrected it.

Technically, this study is small: Only seven scoped names are defined in the code of the solitary Jak-like feature module; only nine names can be generated additionally with five RFMs. The compressed decision trees had a height of just two nodes in average and just five nodes at maximum. The verifier created the trees and evaluated them in 0.2 seconds and thus outperformed our manual attempt by far.[10]

Large-scale Jak-like feature modules. We studied a version of the large-scale Eclipse module ’workbench.texteditor’ (∼16K lines of source code; study inspired by [DNMJ08]); we formerly added RFMs to this study in order to analyze the power of RFMs. With this study, we want to test now whether our concept and prototype do exceed memory when Jak-like feature modules are large-scale and create numbers of scoped names. For our study, we moved a release of ’workbench.texteditor’ into a feature module and reimplemented to some extent, with Jak-like feature modules and RFMs, the development steps to create the successor release (the steps were recorded in the module’s versioning system and were described elsewhere [DNMJ08]). With this SPL, we aimed at generating different releases of ’workbench.texteditor’ by selecting features.

This study contained two Jak-like feature modules and three RFMs: One RFM renamed class Levenstein into Levensh̲tein because the versioning system recorded this change; two other RFMs renamed two member variables which both are used to reference objects of the renamed class Levenstein. The feature model defined that the Rename-Field RFMs execute after the Rename-Class RFM; thus, the Rename-Field RFMs reference the member variables to rename using the type name Levensh̲tein (Levenstein no longer exists). As a result, the Rename-Field RFMs require the Rename-Class RFM to execute on their input programs because it creates Levensh̲tein. However, the prototype alerted that the model we used so far for this study (every feature was declared to be independent from others) allowed the Rename-Field RFMs to be executed in a legal SPL product although the Rename-Class RFM did not execute in this product at all; as a result, the input programs for both Rename-Field RFMs might not include the member variables to rename and thus the Rename-Field RFMs could fail. The prototype proposes to add the detected execution constraint to the feature model in order to make it safe.

Technically, the two Jak-like feature modules were large-scale: Together they created 3428 scoped names. The three RFMs created 68 scoped names additionally. The prototype created and evaluated the decision trees in 1.6 seconds and did not run into memory problems.

High number of RFMs. We studied an extended version of the study on the large-scale Eclipse module ’workbench.texteditor’; we extended the study such that it includes a high number of RFMs. With this new study, we want to test whether our concept and prototype terminates in a reasonable time on current computers, when large-scale, Jak-like feature modules and a high number of RFMs must be

[10]We performed the measurements of this chapter on a computer with an Intel Pentium M CPU with 1.5GHz and 512MB RAM running Windows XP SP2. The measurements are meant as hints; for example, they do not include calculations of statistics or calculations which verify safe composition of Jak-like feature modules.

verified. Specifically, we extended the 'workbench.texteditor' study to include 55 RFMs: 27 RFMs renamed class Levenstein, one RFM renamed member variable DefaultCellComputer.levenstein::Levenstein, and 27 RFMs renamed member variable OptimizedCellComputer.levenstein::Levenstein. These RFMs shall create deep decision trees (i.e., with lots of recorded decisions) with different refactorings.

As before, the two Jak-like feature modules can create 3428 scoped names. Our prototype records each name as a root of a new decision tree (i.e., it records 3428 decision trees). The 55 RFMs can create 2538 additional scoped names and the prototype records each of them in the decision trees. In this study, our tree compression became very important: If we had not compressed the 3428 trees, the 57 possible decisions on feature modules (two Jak-like feature modules + 55 RFMs) would have led to trees of which each would be 57 nodes high and of which each would have 2^{56} leaf nodes (each tree is binary and balanced); due to our tree compressions, the prototype had to create and evaluate just 5966 nodes altogether. Interestingly, our compression reduces the average height of all compressed decision trees of this study to be rather shallow (1.53 nodes). Despite the large scale of this study, the prototype created and verified the decision trees in acceptable 6.7 seconds.

The prototype corrected us successfully already in the small-scale proof-of-concept study and so we were not really surprised when the prototype alerted errors for the current large-scale study. The alerted errors led us to missing dependencies between features in the feature model and led us to spelling mistakes in our RFMs (i.e., RFMs were defined to rename member variables which could not exist at all). Still, we did not expect such high number of mistakes as we found with our prototype.

High number of Jak-like feature modules. We verified a study from prior work in which we added a number of RFMs to the GPL. With this study, we want to test whether our concept and prototype work when a number of Jak-like feature modules can create scoped names before RFMs execute.

The GPL version comprised 15 Jak-like feature modules that are followed by 24 RFMs. The Jak-like feature modules generate a number of scoped names and are correlated in complex execution constraints; thus, our manual attempt to verify the RFMs was seriously hampered (the Jak-like feature modules have been verified before with existing safe-composition techniques). Our prototype created the decision trees and evaluated them in 1.5 seconds while we needed minutes to confirm the alerted errors. Several times, we doubted our prototype until we found our own subtle mistakes in the verified RFMs and the verified feature model.

High number of SPL products. We studied an SPL version of the archive-access module ZipMe in which we applied RFMs in order to integrate ZipMe products with incompatible environments. With this study, we want to test whether our concept and prototype work when a high number of programs can be generated from Jak-like feature modules – the studied ZipMe SPL had 128 products. The ZipMe SPL comprised 14 Jak-like feature modules (e.g., *Crc* or *Checksum*) which were followed by one RFM. Our prototype created the decision trees and evaluated them in 0.2 seconds. The prototype did not generate the 128 ZipMe products.

We verified an SPL version of the archive-access module Raroscope in which we applied RFMs in order to integrate Raroscope products with incompatible environments. We again want to test whether our concept and prototype work when a high number of programs can be generated from Jak-like feature modules – the studied Raroscope SPL had 16 products. The Raroscope SPL comprised five Jak-like feature modules (e.g., *Crc* or *OperatingSystem*) which were followed by two RFMs. Our prototype created the decision trees and evaluated them in 0.1 seconds. The prototype again did not generate the 16 Raroscope products.

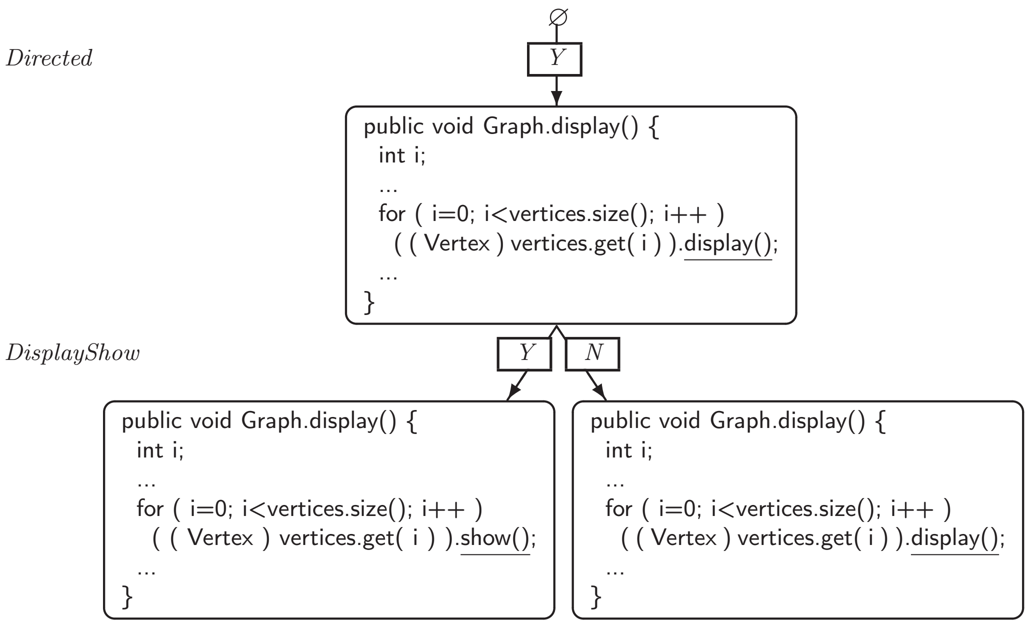
Summary. We evaluated case studies with our prototype and found mistakes which we did not notice before. The studies included SPLs with complex feature models, SPLs with large-scale Jak-like feature modules, SPLs with high numbers of Jak-like feature modules, and SPLs with high numbers of products. The prototype verified all studies in no time while we needed minutes several times to confirm the alerted mistakes. We observed that SPLs become complex rapidly when they include nonmonotonic, nonenumerative feature modules such as RFMs. As a result, even small SPLs became difficult to verify by hand (several times we doubted our prototype until we found the alerted, subtle error). We showed that nonmonotonic, nonenumerative feature modules are beneficial with respect to their use cases (e.g., with respect to integrating modules and with respect to configuring NFPs; cf. Sec. 4.3.1 & Sec. 4.3.2) but to use them in practice we argue: When advanced feature transformations, such as nonmonotonic, nonenumerative feature modules, are used to implement SPLs, programmers need concepts and tools to verify their SPLs; in this chapter, we presented and prototypically implemented such concepts.

5.5. Discussion on Possible Future Extensions

In this section, we discuss possible future extensions to our safe-composition concepts that would allow us to verify preconditions of refactorings toward method bodies (cf. Sec. 5.5.1) and we discuss possible future extensions to our approach that would allow us to verify SPLs without a predefined feature order (cf. Sec. 2.2.2). We do so in order to make our concepts cover all refactorings and presumably even general meta-programming approaches.

5.5.1. Preconditions on Method Bodies

So far, we restricted our focus in order to keep explanations simple: We focused on refactoring-type implementations which have preconditions only on scoped names and relations between scoped names; we did not focus on refactorings which have preconditions on method bodies. Our restricted focus sufficed to verify safe composition for a number of refactoring-type implementations which are important for



```
public void Graph.display() {
  int i;
  ...
  for ( i=0; i<vertices.size(); i++ )
    ( ( Vertex ) vertices.get( i ) ).display();
  ...
}
```

```
public void Graph.display() {
  int i;
  ...
  for ( i=0; i<vertices.size(); i++ )
    ( ( Vertex ) vertices.get( i ) ).show();
  ...
}
```

```
public void Graph.display() {
  int i;
  ...
  for ( i=0; i<vertices.size(); i++ )
    ( ( Vertex ) vertices.get( i ) ).display();
  ...
}
```

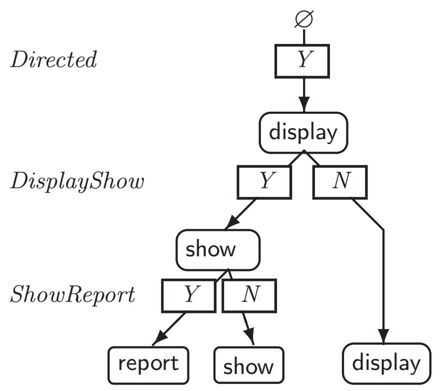
Figure 5.5.: *Tree to record the effect of RFM decisions on the body of a method Graph.display()::void before feature VertexVerteximpl of Fig. 5.3.*

module integration (cf. Sec. 5.3). However, in general, refactoring types may have preconditions on method bodies; for example, a refactoring of type Change Bidirectional Association to Unidirectional requires that no method body references the member variable to remove. We now discuss possible future extensions to our safe-composition approach such that it supports refactoring types that need to verify method bodies.

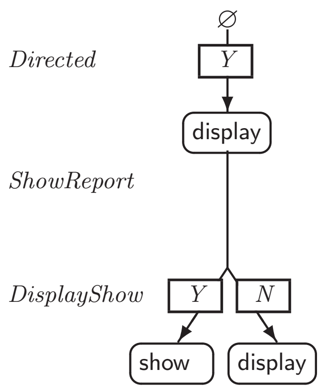
To verify method bodies, we can record method bodies in addition to scoped names in decision trees and their nodes respectively. As a result, refactorings could then also alter the body recorded in a node; as a second result, the trees would branch more frequently than trees which only record scoped names. As an example, we depict the new decision tree for method display of class Graph in Figure 5.5.[11] According to this tree, *DisplayShow* is decisive with respect to the properties of the display node (i.e., the node records a call to the Vertex method display and *DisplayShow* alters this call); while *DisplayShow* was not decisive for nodes of the tree of Graph.display()::void before, this tree must branch for *DisplayShow* as soon as method bodies are considered. Note that the trees recording method bodies would also branch for refinements of Jak-like feature modules because Jak-like feature modules alter method bodies.

To make the extension work, a verification tool should relate references of method bodies to the nodes of the referenced pieces of code. These references will allow to

[11]We slightly adapted the syntax of the method signature in Fig. 5.5 by adding the scoped name of the host class to the method name.



(a) *Decision tree with feature order (ShowReport • (DisplayShow • Directed)).*



(b) *Decision tree with feature order (DisplayShow • (ShowReport • Directed)).*

Figure 5.6.: *Influence of the feature order on the safe-composition approach.*

determine whether a method body and its tree node respectively must be updated (extended by child nodes) on behalf of a change on some other piece of code.

Decision trees, which record method bodies, can be integrated smoothly with the safe-composition concepts we presented in Section 5.3: Suppose that a transformation has preconditions on method bodies, then we could find in the decision trees those leaf nodes which represent methods of which the bodies satisfy the preconditions; we could compute with the decision trees the paths from respective leaf nodes to the empty program; we could transform the decisions of these paths to propositional constraints; finally, we could verify these constraints with a SAT solver in the feature model.

With method bodies in decision trees, we presumably could verify transformations in an SPL which exceed refactorings (e.g., transformations implemented with general meta-programming approaches; cf. Sec. 3.3, p. 26). What remains to be done to cover general meta-programming is presumably to identify the preconditions of individual transformations in these approaches and to implement a template test for them based on the decision trees.

5.5.2. The Influence of the Feature Order

We defined concepts of different functions and steps, which a tool can work out, in order to verify safe composition for an SPL with nonmonotonic, nonenumerative feature modules. One of these functions, the c function[12], relies on a total order of features of an SPL. While a total order might be beneficial (cf. Sec. 2.2.2, p. 13), SPL approaches might exist with no such total feature order. As a result, we now analyze how we could verify safe composition for SPLs with no total feature order.

[12]Function c calculates feature-module sequences that adhere to the feature order and make a scoped name exist in an input program of a feature module (cf. Sec. 5.3.1, p. 68).

When there is no total feature order, a sequence of feature modules may be reordered legally (i.e., a reordered sequence of feature modules still can represent an SPL product that is legal to a feature model). When the feature order changes, our approach does not change conceptually but its result may change; the reason is that if the feature modules are iterated in the different orders, different decision trees are recorded in the course of function c. As an example, we show two decision trees in Figure 5.6; these trees record the effects of decisions on executing the feature modules *Directed*, *DisplayShow*, and *ShowReport* on the method Vertex.display()::void of Figure 5.3: The tree of Figure 5.6a would be created in the course of c when the feature order is (*ShowReport* • (*DisplayShow* • *Directed*)); the tree of Figure 5.6b would be created in the course of c when the feature order is (*DisplayShow* • (*ShowReport* • *Directed*)). We can use both trees to calculate a set of feature-module sequences which make the display method exist in an input program with a certain scoped name (note that both sets of sequences adhere to different feature orders), and use each respective set to generate an execution constraint that we can validate with respect to a feature model as before. For example, when the tree of Figure 5.6a is created in the course of c (i.e., feature order is (*ShowReport*•(*DisplayShow*•*Directed*))), then the calculated set of feature-module sequences, which make a method Vertex.report()::void exist after executing *Directed*, *DisplayShow*, and *ShowReport*, is {(*ShowReport* • (*DisplayShow* • *Directed*))}; when the tree of Figure 5.6b is created in the course of c (i.e., feature order is (*DisplayShow*•(*ShowReport*•*Directed*))), the same set of feature-module sequences is empty (because there is no leaf report).

If there is no total feature order, we propose to start our safe-composition approach repeatedly, each time with a new feature order – there are $n!$ different feature orders for n features. The SPL then would be regarded as safe when it is regarded as safe for every feature order. Note that the absolute number of starts of our safe-composition approach might be very high.

5.6. Summary

In this section, we analyzed the complexity of managing RFMs in SPLs. We presented concepts that tools could implement to verify the consistency between the feature model of an SPL and the feature modules of that SPL. Prior work on SPLs discussed concepts that allow tools to verify consistency of feature models with feature modules in which the feature modules (a) monotonically add code or monotonically remove code and (b) enumerate all the pieces of code they transform. We extended and generalized these concepts to also verify safe composition for nonmonotonic, nonenumerative feature modules such as RFMs. We used these concepts to verify refactorings of RFMs of an SPL, which are executed on behalf of features in a feature model. We prototypically implemented the concepts in a tool and evaluated it in a number of case studies. Finally, we discussed possible future extensions to our concepts.

We observed that nonmonotonic, nonenumerative feature modules can increase the complexity of SPLs rapidly. Specifically, subtle mistakes went unnoticed with our manual verification attempts for even small-scale SPLs (with a small amount of code and few RFMs). We conclude that programmers depend on concepts and prototypes as we presented in this chapter to implement and manage nonmonotonic, nonenumerative feature modules in SPLs.

6. Practical Issues of Using RFMs

In Chapter 3, we reported on the module-scalability dilemma; in Chapter 4, we tackled this dilemma with RFMs and showed benefits and limitations of the approach; in Chapter 5, we observed that – though we were able to prototypically automate a test to verify safe composition of SPLs that involve RFMs – we were hardly able to verify safe composition of such SPLs by hand. So is the RFM concept worth to be analyzed in further detail?

We argue that the RFM concept is worth to be analyzed in further detail because only the *combination* of Jak-like feature modules and RFMs supports the integrated, feature-driven configuration of a module's functionality and structure. This configuration helped us in studies to reuse modules; modules which could not have been reused as is before. We further argue that the complexity of managing SPLs that involve RFMs can be reduced by tools (cf. Chap. 5). This discussion boils down to the worthiness of code reuse and we cannot give a final answer here but can only evaluate the new concept as best as possible in every possible respect. In the end, we also may consider RFMs to show boundaries for the capabilities of program transformations that should be used in SPLs.

In this chapter, we analyze whether common tasks in SPL development become complex due to RFMs, and to what extent we can support programmers with tools for these tasks. These topics have been analyzed in Diploma theses and Master theses that were conducted and supervised in the course of this thesis. In Section 6.1, we discuss how the functionality of SPL products can be corrected when they have been generated using RFMs. In Section 6.2, we discuss how SPL products can be generated faster when they have been generated using RFMs. In Section 6.3, we discuss problems of using RFMs to refactor SPL products, which include artifacts of different languages.[1]

[1]Section 6.1 is based on the Diploma thesis of Martin Sturm [Stu10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [KS10a, KS10b]. Section 6.2 is based on and extends the Master thesis of Liang Liang [Lia10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [KLS10a, KLS10b]. Section 6.3 is based on the Diploma thesis of Hagen Schink [Sch10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [SK10, SKSL11].

6.1. Correcting Errors in the Functionality of SPL Products

Section 6.1 is based on the Diploma thesis of Martin Sturm [Stu10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [KS10a, KS10b].

In prior research, we and others observed that incorrect code of SPLs and SPL products was difficult to detect and correct [AKGL10, DCB09, KBK09, TBKC07]. The according approaches detect noncompilable SPL products, but these approaches do neither detect nor correct errors in functionality.

In this section, we first compare approaches (a) for how programmers can detect incorrect code in SPLs and SPL products (we call this *error detection*), and (b) for how programmers can correct incorrect code (we call this *error correction*). We compare error detection and error correction when they start (a) at the *level of the feature modules* or (b) at the *level of a single SPL product*. We conclude that starting at the *level of the feature modules* can be inappropriate, and that SPL programmers need techniques to start at the *level of a single SPL product*. Finally, we present a new approach which automatically propagates error corrections from a product of an SPL to the feature modules of this SPL. We prototypically implement the presented concepts and demonstrate them feasible.

6.1.1. Comparison of Approaches for Detecting & Correcting Errors

We will exemplify the following discussions with the code of Figure 6.1a. In this figure, we review the slightly modified feature modules *Directed*, *ShortestPath*, and *DisplayShow* of our GPL running example (we introduced an error on purpose). In Figure 6.1b, we review the product which the feature modules of Figure 6.1a generate. The product is supposed to print the count of invocations of `Vertex.display()::void` but it prints zeros instead. The reason is that Line 7 of Figure 6.1b is in error; to correct the product, we must replace Line 7 by "`counter=counter+1;`". Ultimately, however, this correction must occur in the feature modules of Figure 6.1a. To correct the SPL product, we can (a) start at the *level of the feature modules* (cf. Fig. 6.1a) and generate afterwards the corrected product, or (b) start at the *level of the SPL product* (cf. Fig. 6.1b) and propagate afterwards the correction to the feature modules. We found strengths and weaknesses for both approaches with respect to *detecting incorrect code*, *distraction from correcting the product in error*, *SPL complexity*, and the *propagation of changes between levels of code abstraction*.

Detecting Incorrect Code

When a programmer detects incorrect code, the pieces of code that are displayed to him/her in consecutive steps should be reasonable according to code at the level he/she inspects. During error detection, values of variables should be available for inspection.

Feature module *Directed*

```
public class Vertex {
 public String name;
 public void display(){
  ...
  System.out.println();
 } }
```

Feature module *ShortestPath*

```
public refines class Vertex {
 private String predecessor;
 private int dweight;
 int counter=0;
 public void count(){
  counter = counter/1;
  System.out.print(counter);
 }
 public void display() {
    System.out.print( "Pred " + predecessor + "
      DWeight " + dweight + " " );
    count();
    Super.display();
  } }
```

Feature module *DisplayShow*

```
Rename method Vertex.display()::void into show
```

(a) Feature modules with an error.

```
public class Vertex {
   public String name;
   private String predecessor;
   private int dweight;
   int counter=0;
  public void count(){
   counter = counter/1;
   System.out.print(counter);
  }
   public void show() {
     System.out.print( "Pred " + predecessor + "
       DWeight " + dweight + " " );
     count();
     ...
     System.out.println();
   } }
```

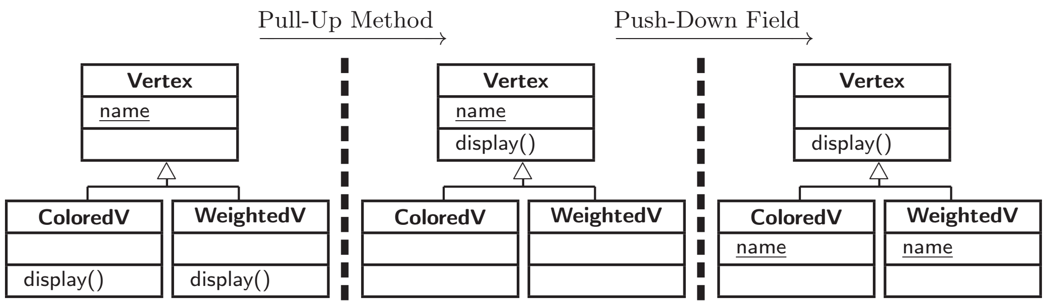
(b) Program generated from all feature modules of Fig. 6.1a.

Figure 6.1.: *Feature modules with an error and the program they generate [adapted from KS10b].*

When programmers start at the level of one SPL product, the pieces of code that are displayed to a programmer in consecutive steps are reasonable according to the code at the level of one SPL product (the displayed code actually is the executed code). The programmer can inspect values of variables.

When programmers start at the level of the feature modules, there might be no code to display to programmers at all: For example, an Encapsulate-Field RFM, creates get-access and set-access methods but does not include these methods; however, skipping this code at the level of the feature modules, would be error-prone [Fai98]. If such access methods further got refined, even more code would be in question.

When programmers start at the level of the feature modules, incorrect code might be displayed to programmers: For example, in Figure 6.2, we depict the input program (cf. Fig. 6.2a) and the generated program (cf. Fig. 6.2b) of a Pull-Up-Method RFM. This RFM merges the method ColoredV.display()::void with method WeightedV.display()::void to generate method Vertex.display()::void. In the generated program, commonly only one method can be encoded to represent Vertex.display()::void at the level of the feature modules (either the ColoredV or the WeightedV method).



(a) *Original program.* (b) *Generated program #1.* (c) *Generated program #2.*

Figure 6.2.: *Refactorings that merge and multiplex code [adapted from KS10b].*

This encoding however can be incorrect according to the method called at the level of the feature modules (ColoredV.display()::void or WeightedV.display()::void). For example, if ColoredV.display()::void is encoded but WeightedV.display()::void is called, ColoredV.display()::void is displayed though it should not.

When programmers start at the level of the feature modules, breakpoints might not work properly: For example, if ColoredV.display()::void of Figure 6.2a is encoded to be displayed when Vertex.display()::void is executed, then a breakpoint set to ColoredV.display()::void would be associated to Vertex.display()::void and thus can interrupt WeightedV.display()::void, though it should not; a breakpoint set in this situation to WeightedV.display()::void would not be associated to the generated, executed method and thus never interrupts the product though it should.

When programmers start at the level of the feature modules, values of variables cannot be inspected, generally: For example, in Figure 6.2, we depict the input program (cf. Fig. 6.2b) and the generated program (cf. Fig. 6.2c) of a Push-Down-Field RFM; this RFM multiplexes the static member variable Vertex.name::String in the classes ColoredV and WeightedV.[2] At the level of the feature modules, the value of Vertex.name::String must be merged from the values of all variables generated from Vertex.name::String (ColoredV.name::String and WeightedV.name::String in Fig. 6.2c) – this merge however is not always possible.

Approaches exist that *could* relate product code to feature-module code [AT96, Hen82, Zel83], but these approaches rely on that conditional breakpoints and path analyses can be compiled into binaries. While this is possible for some languages, it requires high effort to implement supportive tools for every new language.

Distraction from Correcting the Product in Error

In the first place, programmers should focus on the product which is in error. For that, code of the product which is in error should be displayed cohesively and other code should be hidden. As a result, the programmer is not distracted from correcting the product in error, can understand this product more easily, and can correct it more

[2]We modified the field name with static here to ease explanations.

easily; for example, dangling references can be avoided more easily because all code, which is available in the product, is displayed and all code, which is not available in the product, is hidden. Just in the second place, programmers should focus on other products. Finally, the code displayed to a programmer should be in a language, the programmer is familiar with.

When programmers start at the level of one SPL product, the code of the analyzed SPL product is displayed cohesively to a programmer. Other code is hidden at that time. After programmers corrected the product, they can focus on how to integrate the corrections with feature modules. The integration of corrections can be supported by tools; for example, tools can verify that the correction of feature modules does not put other products of this SPL syntactically in error.

For Jak-like feature modules and RFMs, the displayed, generated code is in a language (Java) which is very similar to the language the programmer used to implement the SPL (Jak and RFM).

When programmers start at the level of the feature modules, code of all feature modules of the SPL is displayed to the programmers, even of feature modules that do not contribute to the product in error (i.e., code that does not necessarily contribute to the error). Furthermore, the code of a product, which is in error, is not cohesive and cannot be displayed cohesively but is scattered across feature modules. Note that if tools would visualize generated code, they switch to the approach *correct at the level of one SPL product.*

For Jak-like feature modules and RFMs, the displayed code is in the language which the programmer used to implement the SPL.

SPL Complexity

Bounded quantification is a guideline for how programmers should write feature modules; following this guideline promises to simplify SPLs.[3] To not increase complexity of an SPL, bounded quantification should hold before error correction and after; that is, no programmer/tool should correct a method, field, or class in a feature module such that the program generated by this corrected feature module has a dangling reference. Programmers should be advised accordingly (before changing a feature module), which feature module they should correct (we call this feature module *target*) and how they should correct it.

When programmers start at the level of one SPL product, a tool can support programmers in all three steps: (a) identify corrected pieces of code in the SPL product, (b) advise which feature module to use as a target, and (c) verify that propagating a piece to its target does not break bounded quantification in *any* SPL product. That is, programmers can be supported before a correction (which already is implemented in the SPL product) is made to the feature modules, to decide, which feature module they should correct best (e.g., such that bounded quantification is not broken in *any* SPL product).

[3]Bounded quantification is a guideline according to which a feature module should not generate code with dangling references (cf. Sec. 2.2.2, p. 14).

When programmers start at the level of the feature modules, a programmer *cannot* be supported before a correction is made to the feature modules (i.e., before implementing it), to decide, which feature module he/she should correct best (e.g., such that bounded quantification is not broken in *any* SPL product). After a correction has been implemented, the programmer could be advised in restructuring the feature modules (if possible).

Propagation of Changes Between Levels of Code Abstraction

The tools, which support to integrate corrections with feature modules, should never get stuck in any situation.

When programmers start at the level of one SPL product, a tool should propagate each piece of corrected code from the product to a target (feature module to correct). For that, this tool must invert – in the corrected SPL product – all feature modules in reverse feature order, which followed the target during the generation of the corrected product: To invert a Jak-like feature module, the tool must remove the created code in the SPL product; to invert an RFM, the tool must execute a refactoring in the SPL product which inverts this RFM.

Certain corrections to a product may preclude a tool to invert feature modules that generated the product. For example, when a programmer created a method Vertex.display()::void in the code of Figure 6.1b, no tool can smoothly invert *DisplayShow* ("Rename method Vertex.display()::void into show") in the corrected SPL product (i.e., Vertex.show()::void cannot be renamed into display); if a tool would invert *DisplayShow* smoothly, two pieces of code with equal scoped names would occur in future *DisplayShow* input programs though this is not allowed in most languages (cf. Sec. 2.3, p. 16). For that, there must be a fall-back strategy – for Jak-like feature modules and RFMs, such strategy exists and is discussed later.

When programmers start at the level of the feature modules, they do not need a tool to propagate corrections between levels of code abstractions – they simply regenerate the product. However, if an error occurs in the generation process after the programmers corrected the feature modules, these programmers got stuck.

Summary

When programmers start to correct an SPL at the level of the feature modules, incorrect code may be displayed to them, code that is scattered across feature modules may be displayed to them, and only weak tool support can be provided to them (e.g., to decide which feature module to change). On the one hand, the approach of starting to correct SPLs at the level of a single SPL product avoids these problems; on the other hand, this approach calls for a fall-back strategy. Noteworthy, *none* of the above approaches can guarantee that all SPL products work *as intended* after correcting the feature modules.

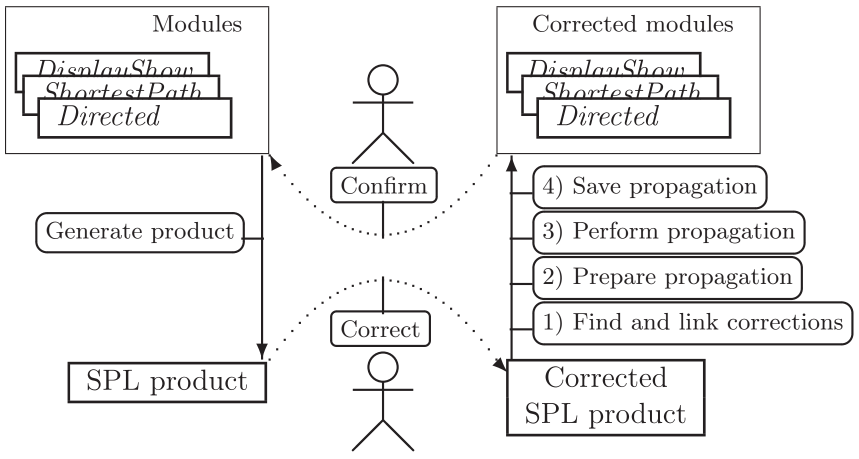


Figure 6.3.: *Use case to correct SPL products and to update feature modules [extended and adapted from KS10b].*

6.1.2. Propagation of Error Corrections to Feature Modules

Some researchers argue that programmers should detect errors and correct errors of an SPL product at the level of the feature modules (there: high-level code of an unoptimized, stand-alone program) [CIBR00, HCU92, WGM08]; from our analysis and in line with others [AT96, CE00, EBLSP10, Fai98, FNP97, IKI04] we argue that programmers should also be allowed to detect errors of a single SPL product at the level of this single product. We show now how programmers could be supported by tools when (after error detection) they start to *correct* an SPL product at the level of this single product.

Conceptual Process

When the programmer detected incorrect code of a single SPL product and corrected this code at the level of this single product, he/she could trigger a tool which propagates his/her correction to the feature modules. This propagation tool should *find and link corrected pieces code*, should *prepare the propagation*, should *perform the propagation*, and should *save the propagation*. We visualize this process in Figure 6.3.

Find and link corrected pieces of code. The propagation tool should first find all the corrected pieces of code in the SPL product (e.g., changes made to individual methods). The tool can find these pieces by comparing the corrected SPL product with the old and incorrect version of the same product. The tool then should link each piece of corrected code to a scoped name (allows us later to compute *good* targets). For example, in Figure 6.1b, the propagation tool should find the correction in Line 7 (which we corrected to be counter=counter+1;) and should link it to the scoped name Vertex.count()::void.

Table 6.1.: *Index that relates scoped names of products to feature modules of Figure 6.1b [adapted from KS10b].*

Index key	**Index value**
Vertex.show()::void	[*Directed:*Vertex.display()::void, *ShortestPath:*Vertex.display()::void]
Vertex.name::String	[*Directed:*Vertex.name::String]
Vertex.predecessor::String	[*ShortestPath:*Vertex.predecessor::String]
Vertex.dweight::int	[*ShortestPath:*Vertex.dweight::int]
Vertex.count()::void	[*ShortestPath:*Vertex.count()::void]
Vertex	[*ShortestPath:*Vertex]

Prepare the propagation. The propagation tool should create an index data structure and a transformation-history data structure to compute good targets for each piece of corrected code later. The tool should create an index data structure which maps each piece of code x of the SPL product to those pieces of code in feature modules which create and include (parts of) x. The index should map x to an empty value when an RFM created x (then there is no code in the feature modules which includes x). For example, we show a sample index as we propose it for the product of Figure 6.1b in Table 6.1. A key in this index is a piece of code (represented by a scoped name) of the SPL product; a value in this index is a list of pieces of code in feature modules that create a piece of code in the analyzed product. The index of Table 6.1 tells us that the code of Vertex.show()::void in the SPL product was created by and is included in the pieces of code Vertex.display()::void of the feature modules *Directed* and *ShortestPath*.

The propagation tool should create a transformation-history data structure, which records the names of all executed feature modules in the execution order of these feature modules. For example, the transformation history for the product of Figure 6.1b should record that *DisplayShow* executed after *Directed* and *ShortestPath*. The history data structure should also record the input program of every executed feature module together with an own index structure for all of these programs.

Perform propagation. The propagation tool should calculate a *good* target for each piece of corrected code. The tool then should propagate each piece to its target by inverting all feature modules on the piece which follow the target in the transformation history. The tool could calculate a good target for a piece of corrected code in three steps: *First*, the tool could detect, which piece of code in the feature modules creates and includes the corrected piece of code, or which feature module created most of the pieces of code the corrected piece relates to – respective feature modules are considered to be good targets. *Second*, the tool could verify that every feature module, which should be inverted in the course of the propagation, can still be executed correctly after the propagation. The tool additionally could verify safe composition of the corrected SPL (i.e., that every product of the corrected SPL can be generated without error; cf. Chap. 5). The tool could adjust the target if needed.

If the tool calculates that one feature module should be inverted to propagate a corrected piece of code but cannot be inverted on that corrected piece of code, the tool could apply a *fall-back strategy* such as: Create a Jak-like feature module which follows the noninvertible feature module according to the feature order and let the new feature module be the new target. This new feature module would be executed at least during future generations of the corrected product and would replace therein the erroneous code. *Third*, the tool could verify that propagating a piece of corrected code to its target does not break bounded quantification for any product. The tool could adjust the target if needed.

To exemplify the propagation of a corrected piece of code, we reconsider the correction in Figure 6.1b (replace Line 7 by counter=counter+1;). The correction involves only one piece of code and this piece is linked to the scoped name Vertex.count()::void. Then the tool could compute a target in three steps: First, the tool could analyze with the index for the code of Figure 6.1b (cf. Tab. 6.1) that the method Vertex.count()::void has been created and is included in feature module *ShortestPath*; thus, *ShortestPath* is the target for count. Second, the tool could verify that every feature module which follows *ShortestPath* still executes correctly after the tool propagated the corrected piece of code to *ShortestPath*. Third, the tool could verify that a corrected count in *ShortestPath* does not break bounded quantification. In this example, the tool would not need to adjust the target. After the tool would have computed the target, the tool could invert *DisplayShow* on the piece of corrected code (nothing changes) and would retrieve the corrected piece of code to insert into the feature module *ShortestPath*.

When a piece of corrected code does not reference other pieces of code and the scoped name relates to an empty value in the index (i.e., the piece got created by an RFM) then the tool should propagate the piece along the reverse sequence of feature modules (which are recorded by name in the transformation history) without target. This propagation should stop as soon as the tool recognizes that the feature module it inverts created the incorrect piece of code in the first place; the tool then can provide a new target for the propagated piece of code. For example, if *DisplayShow* would have created the corrected Vertex.count()::void in Figure 6.1b, then Vertex.count()::void would exist in the index but would map to an empty value. The tool then would invert all feature modules *DisplayShow* to *Directed* in their reverse feature order in the code of Vertex.count()::void. As soon as the tool would invert *DisplayShow*, the tool would detect that *DisplayShow* created the incorrect Vertex.count()::void in the first place; the tool then could compute a new target for the corrected method count. For example, the tool then could define the target to be a Jak-like feature module, which follows the RFM (according to the feature order) that created the incorrect count method. Note that if *DisplayShow* would have merged the corrected method Vertex.count()::void from multiple pieces of code in the first place, then the tool which inverts *DisplayShow* could detect that *DisplayShow* created Vertex.count()::void in the first place, and could define multiple new targets for the correction.

Save propagation. The propagation tool might compute that a certain feature module is a good target for a piece of corrected code; a programmer however might consider a different feature module to be a better target for the same piece. The propagation tool thus should always ask the programmer to confirm the target and the correction proposal for each piece.

Unsupported Error Corrections & Feature Module Capabilities

The capabilities of feature modules limit the corrections which a tool can propagate from a product of an SPL to the feature modules of that SPL. With respect to the Jak language, no fall-back strategy seems available:

Jak-like feature modules written in Jak cannot replace constructors in their input programs [Sof08]. However, replacing an incorrect piece of code in a new feature module was our fall-back strategy when we could not propagate corrections. We thus need and propose a second fall-back strategy (based on the advice from the Jak documentation [Sof08]): Extract constructor bodies into methods and propagate corrections to these methods instead.

Jak-like feature modules written in Jak, as well as RFMs cannot remove code in their input programs without a replacement. As a result, the propagation tool may fail when it cannot propagate a correction, in which a programmer removed a piece of code from the SPL product.

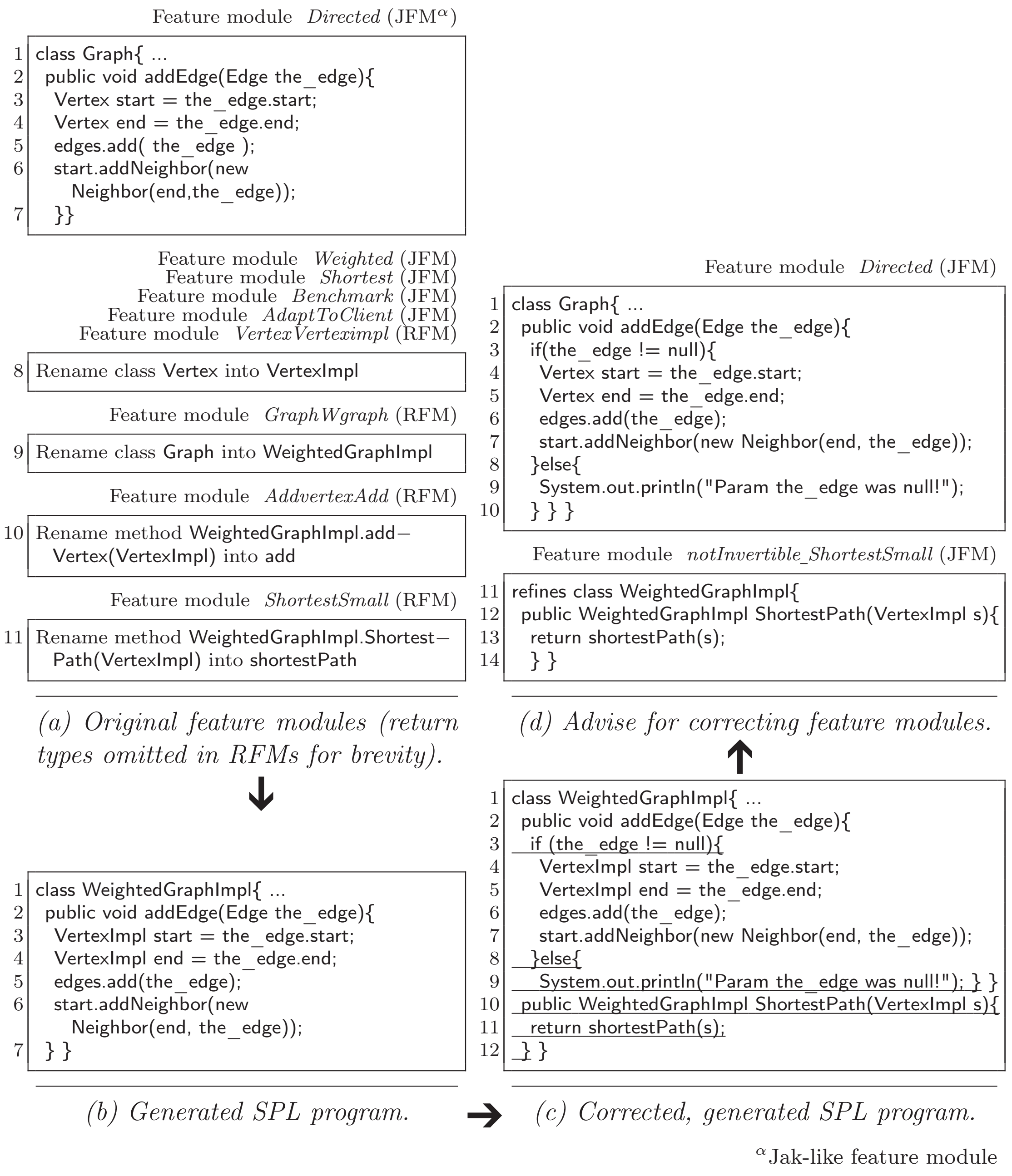
In general, feature modules of SPLs must be invertible to propagate corrections from an SPL product toward the feature modules. Many program transformations used in SPLs can be inverted [BSR04, KBA09, KHH$^+$01], but certainly not all. When a transformation cannot be inverted our fall-back strategy should be used.

A tool, which propagates corrections from a product of an SPL to the feature modules of that SPL, will depend on the tool, which was used to generate the product in the first place. If different composer tools generate different products for the same feature modules (common for FOP [Bat06]), the propagation tool must differ with respect to the used composer tool. Note that different composer tools add complexity to every possible solution.

6.1.3. Prototype & Demonstration

We prototypically implemented the index concept and the transformation-history concept for SPLs with Jak-like feature modules and RFMs.[4] We used this prototype in a demonstrating example. Our prototype finds pieces of corrected code in the SPL product and links them to scoped names (Step *Find and link corrected pieces of code* in Sec. 6.1.2), creates an index data structure and a transformation-history data structure (Step *Prepare the propagation*), propagates each piece of corrected code to

[4]For example, the prototype so far does support only one composer tool, does not keep complete input programs of feature modules (thus, it does neither support so far to detect name capture nor to invert RFMs that merge code), does not guarantee safe composition for the corrected SPL, and maintains only one index structure at all.



Feature module *Directed* (JFM$^{\alpha}$)

```
class Graph{ ...
 public void addEdge(Edge the_edge){
  Vertex start = the_edge.start;
  Vertex end = the_edge.end;
  edges.add( the_edge );
  start.addNeighbor(new
    Neighbor(end,the_edge));
  }}
```

Feature module *Weighted* (JFM)
Feature module *Shortest* (JFM)
Feature module *Benchmark* (JFM)
Feature module *AdaptToClient* (JFM)
Feature module *VertexVerteximpl* (RFM)

Rename class Vertex into VertexImpl

Feature module *GraphWgraph* (RFM)

Rename class Graph into WeightedGraphImpl

Feature module *AddvertexAdd* (RFM)

Rename method WeightedGraphImpl.add−Vertex(VertexImpl) into add

Feature module *ShortestSmall* (RFM)

Rename method WeightedGraphImpl.Shortest−Path(VertexImpl) into shortestPath

(a) Original feature modules (return types omitted in RFMs for brevity).

```
class WeightedGraphImpl{ ...
 public void addEdge(Edge the_edge){
  VertexImpl start = the_edge.start;
  VertexImpl end = the_edge.end;
  edges.add(the_edge);
  start.addNeighbor(new
    Neighbor(end, the_edge));
 } }
```

(b) Generated SPL program.

```
class WeightedGraphImpl{ ...
 public void addEdge(Edge the_edge){
  if (the_edge != null){
   VertexImpl start = the_edge.start;
   VertexImpl end = the_edge.end;
   edges.add(the_edge);
   start.addNeighbor(new Neighbor(end, the_edge));
  }else{
   System.out.println("Param the_edge was null!"); } }
 public WeightedGraphImpl ShortestPath(VertexImpl s){
  return shortestPath(s);
 } }
```

(c) Corrected, generated SPL program.

Feature module *Directed* (JFM)

```
class Graph{ ...
 public void addEdge(Edge the_edge){
  if(the_edge != null){
   Vertex start = the_edge.start;
   Vertex end = the_edge.end;
   edges.add(the_edge);
   start.addNeighbor(new Neighbor(end, the_edge));
  }else{
   System.out.println("Param the_edge was null!");
  } } }
```

Feature module *notInvertible_ShortestSmall* (JFM)

```
refines class WeightedGraphImpl{
 public WeightedGraphImpl ShortestPath(VertexImpl s){
  return shortestPath(s);
  } }
```

(d) Advise for correcting feature modules.

$^{\alpha}$Jak-like feature module

Figure 6.4.: *Error-correction process for a GPL class* WeightedGraphImpl *(formerly* Graph*) [from KS10b].*

a good target (Step *Perform propagation*), and saves the corrected Jak-like feature modules separately (Step *Save propagation*).

Demonstrating example. To demonstrate our prototype, we use a version of the GPL from prior work (cf. Sec. 4.3.1) to which we added RFMs.[5] We generated a compilable GPL product from five Jak-like feature modules and four RFMs. We list the names of these nine feature modules top-down in Figure 6.4a in feature order together with relevant excerpts of these modules. As we could not detect mistakes in the GPL product, we applied *changes* to the product's classes WeightedGraphImpl and VertexImpl; changes which cover interesting cases during propagation. For homogeneity, we call our changes *corrections*.

We demonstrate in Figures 6.4a-d, how we corrected code in the class WeightedGraphImpl (formerly Graph; cf. Fig. 6.4a) in the product and how we propagated the corrected code to the feature modules. In Figure 6.4a, we list the GPL feature modules (and relevant excerpts of them), which we executed to generate the studied product. In Figure 6.4b, we show (relevant excerpts of) the studied product.

We observe the product of Figure 6.4b to malfunction, detect its incorrect code, and correct it (cf. Fig. 6.4c, we underlined the pieces of corrected code); then we start our prototype to propagate the correction. The prototype compares an old and incorrect version of the GPL product (cf. Fig. 6.4b) with its corrected counterpart (cf. Fig. 6.4c). The prototype finds three pieces of corrected code, links two of them (Lines 3 and 8-9) to the scoped name WeightedGraphImpl.addEdge(Edge)::void, and links one of them (Lines 10-12) to the scoped name WeightedGraphImpl.ShortestPath(VertexImpl)::WeightedGraphImpl.

First, the prototype propagates the corrections in method WeightedGraphImpl.addEdge(Edge)::void. The prototype detects with the index data structure that (a) WeightedGraphImpl.addEdge(Edge)::void got altered and not added during error correction (i.e., index key WeightedGraphImpl.addEdge(Edge)::void exists) and that (b) the incorrect WeightedGraphImpl.addEdge(Edge)::void got created and is included in the feature module *Directed*. So, *Directed* is target for the correction in addEdge. The prototype verifies using the transformation history that the RFMs, which follow *Directed*, can be inverted in the code of addEdge (Jak-like feature modules can always be inverted).[6] The prototype verifies using the index that a corrected addEdge in *Directed* does not break bounded quantification. Finally, the prototype propagates the correction and saves the corrected feature module *Directed* separately (cf. Fig. 6.4d). Once the programmer confirmed the target and the correction proposal, the corrected feature module could replace the original feature module (replacement not automated so far). Note that the pieces of corrected code of Line 3 and Lines 8-9 of Figure 6.4c are propagated together because they affect the same piece of code.

Second, the prototype propagates the correction in method WeightedGraphImpl.ShortestPath(VertexImpl)::WeightedGraphImpl. The prototype detects with

[5]We pruned the GPL version to fit the current capabilities of our prototype; for example, we removed RFMs when they did not implement refactorings of type Rename Class or Rename Method.

[6]The prototype so far depends on a composer tool which generates a method for every method refinement [Bat06]. Inverting a refinement thus means to remove its according method. Other composer tools might require more sophisticated mechanisms.

```
 1 class VertexImpl{ ...
 2  private boolean displayed = false;
 3  public void display() {
 4   System.out.print("Pred " + predecessor + " DWeight " + dweight + " ");
 5   display$$eval$outWeighted$GG();
 6   this.displayed = true; }
 7  public boolean wasDisplayed(){
 8   return displayed; }
 9  public VertexImpl assignName(String name) {
10   this.name = name;
11   if(this.wasDisplayed()){
12    System.out.println("was already displayed!"); }
13   return (VertexImpl)this;
14  } }
```

Figure 6.5.: *Corrected GPL class Vertex/VertexImpl [from KS10b].*

the index data structure that ShortestPath got created during error correction (i.e., index key WeightedGraphImpl.ShortestPath(VertexImpl)::WeightedGraphImpl does not exist). After detecting this, the prototype detects that (a) ShortestPath solely references method shortestPath in the same class, and that (b) this referenced method shortestPath got created by the feature module *Shortest*. So, *Shortest* is target for the new method ShortestPath. The prototype then checks whether all feature modules, which follow *Shortest*, can be inverted in the code of ShortestPath. RFM *ShortestSmall* would not execute correctly any more in future products when it would be inverted in the code of ShortestPath[7]; that is, the prototype cannot propagate the correction ShortestPath to its target *Shortest*. The prototype implements the fall-back strategy and creates a Jak-like feature module *notInvertible_ShortestSmall* which follows *ShortestSmall* according to the feature order (cf. Fig. 6.4d) and which creates method ShortestPath in future SPL products. Interestingly, *notInvertible_ShortestSmall* refines class WeightedGraphImpl but not Graph because WeightedGraphImpl exists in the input program of *notInvertible_ShortestSmall* (input program generated by *ShortestSmall*) but not Graph.

Finally, the prototype propagates a correction which we made to class VertexImpl (formerly Vertex). We show the corrected class VertexImpl in Figure 6.5 and underlined our correction.[8] First, the prototype detects the corrections in VertexImpl and links them to the scoped names VertexImpl.displayed::boolean, VertexImpl.display()::void, VertexImpl.wasDisplayed()::boolean, and VertexImpl.assignName(String)::VertexImpl. The prototype further detects that the member variable displayed and the method wasDisplayed got created in VertexImpl during error correction, and that methods display and assignName got altered.

The prototype firstly propagates the added member variable displayed. The proto-

[7]After inverting *ShortestSmall*, a future input program of *ShortestSmall* would contain two ShortestPath methods, and this would make *ShortestSmall* fail.

[8]The code in Line 5 of Fig. 6.5 is a possible translation for Super.display(); (cf. Fig. 6.1a, p. 87, Line 18); inlining such calls removes the statement.

type detects that the method display, which was created by Jak-like feature modules, accesses displayed and so, the target for displayed becomes the feature module that lastly creates code in display: *Shortest*.

The prototype secondly propagates the added method wasDisplayed: The prototype detects that wasDisplayed solely accesses displayed and so the target of wasDisplayed becomes the target of displayed: *Shortest*.

The prototype finally calculates targets for the altered methods display and assign-Name. The prototype detects that *Directed* created both methods and thus *Directed* becomes a first target for the corrections in both methods. However, the prototype must adjust the target: The prototype detects that display and assignName reference class members which are created in *Shortest*, displayed and wasDisplayed; to maintain bounded quantification, the prototype adjusts the target for the corrected methods display and assignName: *Shortest*. According to the propagation proposal, *Shortest* will replace display and assignName (which are still created in *Directed*) with correct counterparts in future product generations.

6.1.4. Summary

In this section, we analyzed how programmers can start to detect and correct errors in an SPL product that was generated from Jak-like feature modules and RFMs. We compared two basic approaches: (a) *start at the level of a single SPL product* and (b) *start at the level of the feature modules*. We found strengths and weaknesses for both approaches. We observed that approach (b) is not always better than approach (a); for SPLs implemented with Jak-like feature modules and RFMs, we observed that approach (a) can be beneficial. Finally, we implemented a prototype, which propagates corrections from a generated product of an SPL to the feature modules of this SPL. We demonstrated the prototype using a product of our GPL running example.

6.2. Reducing the Program-Generation Time

Section 6.2 is based on and extends the Master thesis of Liang Liang [Lia10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [KLS10a, KLS10b].

SPL users and SPL programmers generally have different views on an SPL: An SPL user knows features and their meaning but not their implementation in general (e.g., SPL users do not need to know the code of feature modules); in contrast, SPL programmers implement the SPL and, thus, do know the implementation of features. An SPL user prefers fewer selectable features and could even appreciate automated selection of features [Bat05, RS10, WSWN07, ZJ04]; in contrast, an SPL programmer prefers a high number of features because he/she can then satisfy a high number of customers with tailored programs; programmers could even appreciate to reuse individual feature modules to implement different features. That is, SPL programmers prefer to have a small number of feature modules that can be combined in many ways [CE99b].

Based on the different views on SPLs, users may (unknowingly) execute sequences of feature modules, which are longer than necessary to generate a single product and which thus may take a tool longer than necessary to execute. As a simple example, assume that in one SPL product the classes Vertex and Node exist and should swap names; then programmers must define three RFMs (e.g., "Rename class Node into Temp", "Rename class Vertex into Node", and "Rename class Temp into Vertex"). If for a new SPL product only class Vertex exists and should be renamed into Node, then programmers may reuse the already existing sequenced RFMs ("Rename class Node into Temp", "Rename class Temp into Vertex"). The latter sequence of RFMs, however, is longer than necessary and can be replaced by a single refactoring "Rename class Node into Vertex" – this replacement shortens the overall sequence to execute and could reduce the time to generate the product. Finally, sequences of feature modules can also be longer than necessary when SPLs grow large-scale and individual programmers are not in charge of all feature modules.[9]

The problem of suboptimal feature-module sequences corresponds to the problem of suboptimal database queries (discussed in Sec. 7.6). In this section, we lean on optimization techniques of database management systems to optimize (shorten) sequences of RFMs. We discuss the concepts, describe our prototype, and report on the evaluation of that prototype. In our studies, the prototype reduced program-generation time in a number of cases.

6.2.1. Optimizing Refactoring Sequences

Commonly, refactoring tools execute refactorings in two phases (cf. Sec. 2.3, p. 16): A verification phase and a transformation phase. We argue that *both* phases offer potential to optimize sequences of refactorings in order to reduce the time a tool needs to execute the sequences. Researchers showed that verification phases of refactorings in a sequence of refactorings can be omitted when the verified preconditions are known to be satisfied for a refactoring [KK04, Rob99]; after omitting precondition checks, a tool can execute the sequence of refactorings faster.

We show how transformation phases of refactorings in sequences of RFMs can be optimized. For example, when two RFMs follow each other in a sequence, of which the first RFM renames a class Vertex into Temp and of which the second RFM renames Temp into Node, then we replace both refactorings before program generation by a fused refactoring which renames Vertex into Node in one step. For this optimized sequence, we expect performance benefits for the program-generation process because (a) the optimized RFM sequence generates the same program as the unoptimized sequence and (b) the optimized sequence avoids to parse the program multiple times, avoids to detect code relations multiple times, and avoids to transform the program multiple times. The transformation phases of a sequence of refactorings can

[9]Researchers report on long refactoring sequences in which 81 and even 800 refactorings were consecutively executed on stand-alone programs [TB01].

be optimized without any analysis of the program to refactor (we call this *algebraic optimization*) and with an analysis of the program to refactor (we call this *cost-based optimization*).

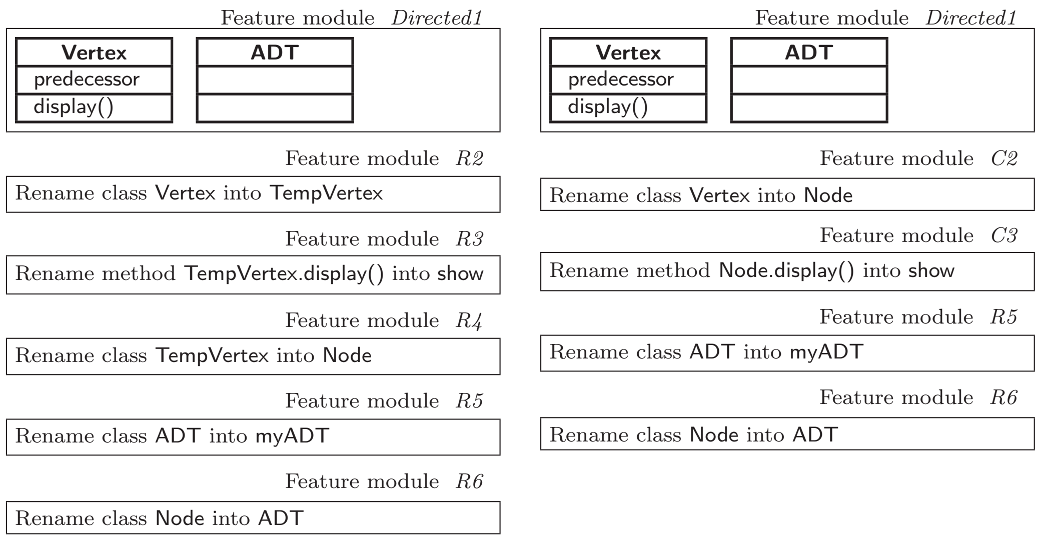
Algebraic Optimization

With algebraic optimization, we aim at fusing *transformation* phases of sequenced refactorings (fusing *verification* phases has been shown before [KK04, Rob99]). To optimize, we propose to first reorder the sequenced refactorings in order to group refactorings with transformation phases detected to be fusible. After this happened, we propose to fuse refactorings based on rules which we define later.

Basic concept. We can iterate a refactoring sequence in order to detect the refactorings with fusible transformation phases. We propose to allow to fuse the transformation phases of two refactorings, when (a) both refactorings follow each other directly in the sequence of refactorings, (b) the refactoring to be executed earlier generates the code which the following refactoring transforms, (c) fusing both refactorings yields either a standard refactoring according to catalogues [Fow99] or the identity transformation (we remove identity transformations as they do not change the generated product), and (d) both are known to execute without errors. We define that the fused refactoring must be a standard refactoring to allow the same fusion rules when the optimized sequence is extended or embedded in another sequence later. To allow nonstandard refactorings as fusion results means to support an infinite number of refactorings in every following optimization step and execution step (nonstandard refactorings have been analyzed elsewhere [KK04]).

In Figure 6.6a, we show a sample sequence of feature modules top-down in their execution order; the Jak-like feature module *Directed1* and a number of RFMs (*R2* to *R6*). In this sequence, we can detect that *R2* (“Rename class Vertex into TempVertex”) can be fused with *R4* (“Rename class TempVertex into Node”) to become a new RFM *C2* (“Rename class Vertex into Node”); we can fuse the refactorings of these RFMs because *R2* generates the code which *R4* transforms (class TempVertex) and because *C2* is a standard refactoring. So far, both refactorings however do not follow each other directly in the sequence of feature modules. We can commute *R4* with its predecessor RFMs to make *R4* the successor of *R2*. Finally, we can fuse *R2* with *R4* – we show the optimization result of Figure 6.6a in Figure 6.6b.

We can detect that *R4* and *R6* can be fused exactly like *R2* and *R4* were fused before (*R4* creates Node lastly which *R6* transforms; fusing *R4* with *R6* would yield the standard refactoring “Rename class TempVertex into ADT”). However, we cannot reorder *R6* to become the successor of *R4* because we cannot commute *R5* with *R6* – if we would commute *R5* with *R6* then *R6* would fail in the resulting sequence because the class ADT would exist in the input program of *R6* though *R6* requires ADT to not exist. To prevent such errors, we propose to detect the dependencies between refactorings (and possibly to record them in a graph data structure; similar to others [Rob99]).


Feature module Directed1
Vertex
predecessor
display()
ADT
Feature module R2
Rename class Vertex into TempVertex
Feature module R3
Rename method TempVertex.display() into show
Feature module R4
Rename class TempVertex into Node
Feature module R5
Rename class ADT into myADT
Feature module R6
Rename class Node into ADT
Feature module Directed1
Vertex
predecessor
display()
ADT
Feature module C2
Rename class Vertex into Node
Feature module C3
Rename method Node.display() into show
Feature module R5
Rename class ADT into myADT
Feature module R6
Rename class Node into ADT

(a) Initial RFM sequence. *(b) Optimized RFM sequence.*

Figure 6.6.: *Initial and optimized sequence of RFMs (return types omitted for brevity) [adapted from KLS10a].*

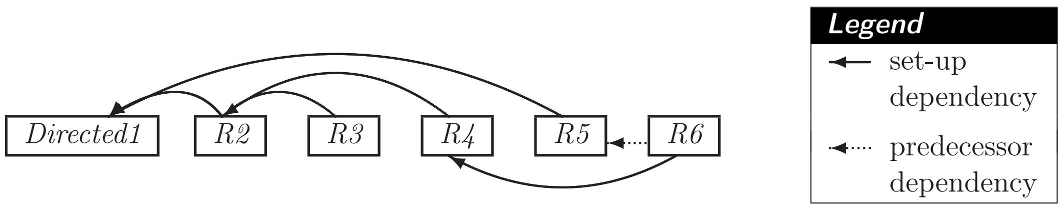


Figure 6.7.: *Conceptual dependency graph for RFMs of Fig. 6.6a [adapted from KLS10a].*

We can analyze the refactorings in their execution order and detect dependencies among them. In particular, we can detect *set-up dependencies* to detect that an RFM requires another feature module to execute on the input program before (to generate a piece of code which the RFM transforms and requires). Further, we can detect *predecessor dependencies* to detect that an RFM requires another feature module to execute before because this other feature module removes a piece of code that the RFM requires to not exist. We show the resulting conceptual dependency graph for the feature modules of Figure 6.6a in Figure 6.7.[10]

With respect to our optimization, set-up dependencies indicate the RFMs of which the parameters possibly must be updated after reordering. We propose to update

[10] *Directed1* generates classes Vertex and ADT which are required by *R2* and *R5*; *R2* generates class TempVertex which is required by *R3* and *R4*; *R4* generates class Node lastly which is required by the following *R6*; *R5* removes ADT which is required by *R6* to not exist.

Table 6.2.: *Fusion rules for optimizing RFM sequences [from KLS10a].*

Preceding RFM	**Follower RFM**	**Fused RFM**
Rename class C_1 into C_2	Rename class C_2 into C_3	Rename class C_1 into C_3
Extract API artifact C_1 into I_1	Rename class I_1 into I_2	Extract API Artifact C_1 into I_2
Rename method M_1 into M_2	Inline method M_2	Inline method M_1
Move class C_1 into C_2	Move class C_2 into C_3	Move class C_1 into C_3
Rename class C_1 into C_2	Collapse hierarchy (C_2,C_3) into C_3	Collapse hierarchy (C_1,C_3) into C_3
Extract class C_1 into C_2	Rename class C_2 into C_3	Extract class C_1 into C_3
Extract method M_1 into M_2	Rename method M_2 into M_3	Extract method M_1 into M_3
Extract class C_1 into C_2	Rename class C_2 into C_3	Extract class C_1 into C_3
Extract class C_1 into C_2	Move class C_2 into C_3	Extract class C_1 into C_3
Extract subclass C_1 into C_2	Rename class C_2 into C_3	Extract subclass C_1 into C_3
Extract subclass C_1 into C_2	Move class C_2 into C_3	Extract subclass C_1 into C_3
Extract superclass C_1 into C_2	Rename class C_2 into C_3	Extract superclass C_1 into C_3
Extract superclass C_1 into C_2	Move class C_2 into C_3	Extract superclass C_1 into C_3
Push-down field F_1 into F_2	Pull-up field F_2 into F_1	∅
Push-down method F_1 into F_2	Pull-up method F_2 into F_1	∅

C represents a class; I represents an API artifact; M represents a method; F represents a field

the parameters of two commuted RFMs when (a) both RFMs depend on the same scoped name and when (b) a parameter value of the former predecessor RFM is affected by the former follower RFM, or vice versa. For example in Figure 6.6a, we can commute *R3* and *R4* to fuse *R4* with *R2*; *R3* and *R4* both depend on TempVertex and a parameter of *R3* (TempVertex.display()) is affected by *R4* – for that, we should update the parameters of *R3* ("Rename method TempVertex.display() into show") to become a new RFM *C3* ("Rename method Node.display() into show"; cf. Fig. 6.6b). Predecessor dependencies indicate which refactorings cannot be commuted at all. For example, we can detect that *R6* cannot be commuted with *R5* at all because we can detect a predecessor dependency between *R5* and *R6*.

Once we are finished with reordering RFMs, we can fuse RFMs that follow each other directly when their refactorings match a pattern we define in Table 6.2 (see [KLS10a, Lia10] for a complete list). For example, when two Rename-Class RFMs follow each other of which the first RFM renames a class and of which the second RFM renames the renamed class a second time, we can fuse both RFMs to become a single Rename-Class RFM which renames the original class to its final name (We can fuse refactorings "Rename class C_1 into C_2" and "Rename class C_2 into C_3" to become "Rename class C_1 into C_3"; cf. Tab. 6.2).

Name capture. Name capture is an error, which can occur when override relations between methods change during refactoring (cf. Sec. 2.3, p. 16). We must prevent name capture when we commute RFMs but we generally do not know whether class members referenced in different RFMs can capture each other or override each other (we execute the RFMs after the optimization but not before). Nevertheless, the program generated from the original feature-module sequence must be equal to the program generated from the optimized feature-module sequence.

To avoid name capture, we cannot commute generally a refactoring "Rename method Vertex.display()::void into show" with a refactoring "Rename method SpVertex.show()::void into display". The reason is that we cannot detect whether Vertex.show()::void (generated by the predecessor refactoring) is overridden by SpVertex.show()::void (transformed by the follower refactoring). Thus, when reordering both refactorings we cannot detect if we must update the refactoring parameters or not. A second example, in which we cannot commute refactorings generally, is when an Extract-API-Artifact RFM follows a Move-Method RFM (or an Inline-Method RFM) and both affect the same class; the reason is that if we would commute both RFMs (i.e., execute the Extract-Interface RFM before the Move-Method RFM) then the Move-Method RFM would fail because only monomorphic methods can be moved [Fow99].

We consider three possible approaches to avoid name capture during our optimization: First, we can detect the override relations between class members in the code, which is generated by the first Jak-like feature module, and can combine these relations with recorded refactoring effects *during* program generation. Before we execute a specific RFM, we can then use our records to calculate whether class members, referenced in this RFM override each other in the input program of this RFM when certain refactorings are known to not apply before the RFM. Second, we can disallow commutation of two RFMs when both RFMs alter methods, both RFMs alter member variables, or when one of these RFMs creates new overriding relations like Extract-API-Artifact RFMs. Thereby, we must only disallow commutation when the altered method names match or the altered member-variable names match. For example, we could commute a refactoring "Rename method SpVertex.show()::void into display" with a refactoring "Rename method SpVertex.setPredecessor()::void into set" without a problem. Third, we can extend RFMs to enumerate the scoped names of all pieces of code they alter (e.g., a Rename-Class RFM could enumerate all methods of which it changes the scoped name or the return-type name). In the last approach, RFM parameters must only be updated after commutation if the sets of scoped names in two RFMs overlap.

Heuristical reordering. Reordering RFM sequences could reduce the time a tool needs to execute the sequences although no transformation phases (and no verification phases) are merged. For example, the time a tool needs to execute an RFM sequence could be reduced when in that sequence an RFM, which renames a member variable (Rename-Field refactoring), follows an RFM, which encapsulates the same member variable (Encapsulate-Field refactoring). When the Rename-Field RFM is executed before the Encapsulate-Field RFM, a refactoring tool must update field references throughout the program for both RFMs; when the Encapsulate-Field RFM is executed before the Rename-Field RFM, the refactoring tool must update only two references for the Rename-Field RFM, and the tool knows the locations of these references (one is in the generated get-access and one is in the generated set-access method). When member-variable names contribute to according access-method names, this particular commutation is not possible because it would alter the generated product.

The time a tool needs to execute an RFM sequence could be reduced when in this sequence an RFM, which alters the access modifier of a method (Hide-Method refactoring[11]), follows an RFM, which renames the same method. When the Rename-Method RFM is executed before the Hide-Method RFM, a refactoring tool must look for references to rename throughout the program. When the Hide-Method RFM is executed before the Rename-Method RFM, the refactoring tool first creates an access modifier for the method; from this modifier, the refactoring tool can predict for the Rename-Method RFM which classes may include references to update. For example, if the Hide-Method RFM modified a method with private[12] before the rename, the refactoring tool could infer that, in order to rename the method, it only must parse, analyze, and update the class, which hosts the method, and its respective superclasses to avoid name capture.

Random reordering. The concepts presented so far cannot detect all fusion potentials; for example, the concepts do not detect that the RFM sequence $(R3 \bullet (R2 \bullet R1))$ can be shortened in which *R1* is “Rename class C_1 into C_2”, *R2* is “Move class C_2 into C_3”, and *R3* is “Rename class C_3 into C_4”. The presented concepts would detect that *R1* cannot be fused with *R2* and that *R2* cannot be fused with *R3* because the resulting RFMs would not implement standard refactorings. However, the concepts do not detect that *R1* can be fused with *R3*; they do not detect this because the earlier executed RFM *R1* does not generate the piece of code which the RFM *R3* transforms later (it actually does generate this code but *R2* prevents that we can detect this from the refactoring parameters).

If the refactoring tool would *at random* commute either *R1* with *R2* or *R2* with *R3*, the above concepts could detect that (reordered) *R1* can be fused with (reordered) *R3*. We envision that refactoring tools clone RFM sequences and commute RFMs at random (if possible) in individual cloned sequences; the randomly reordered sequences could then be analyzed for fusible RFMs again; the shortest, optimized, cloned sequence finally could be executed.

Cost-based Optimization

A tool could reduce the time (other) tools need to execute a sequence of RFMs when the tool would analyze the program which the RFMs shall refactor – the code of this program is commonly generated by the first Jak-like feature module(s).

A tool could analyze inheritance hierarchies and access modifiers in order to identify RFMs which affect distinct parts of a program; when the tool can make these RFMs follow each other directly, the tool then can execute these RFMs in parallel.

[11] A Hide-Method refactoring replaces the access modifier of a method in order to restrict access as much as possible [Fow99].

[12] No code outside a class can reference private members of that class (cf. Sec. 2.1, p. 8).

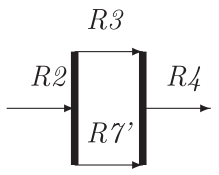


Figure 6.8.: *Process of executing two RFMs in parallel [from KLS10a].*

There are different ways to determine that RFMs affect distinct parts of a program (of which we list only a few): RFMs affect distinct parts of a program when they affect private members of different classes. RFMs affect distinct parts of a program when they reference unmodified members of classes[13] and when these classes are located in different packages and have no superclasses, subclasses, or API artifacts. RFMs affect distinct parts of a program when they affect unmodified classes, these classes (a) are not nested in other classes, (b) have no subclasses, no superclasses, and no API artifacts, and (c) are located in different packages. As an example, assume the RFM *R3* ("Rename method TempVertex.display()::void into show") and an RFM *R7* ("Rename method gpl.Vertex.show()::void into insert"); a refactoring tool could analyze that TempVertex.display()::void and gpl.Vertex.show()::void are unmodified, that the classes TempVertex and gpl.Vertex are in different packages, and that TempVertex and gpl.Vertex have no subclasses, no superclasses, and no API artifacts. Thus, the tool could execute *R3* and *R7* in parallel when it can reorder them to become direct successors in an RFM sequence (cf. Fig. 6.8).

A tool could also analyze inheritance hierarchies and access modifiers in order to identify RFMs which are likely cheap (affect less code). For example, if an RFM is analyzed (a) to transform a piece of code which is modified with private or if an RFM is analyzed (b) to transform a piece of code that is modified with protected and located inside a shallow inheritance hierarchy, this RFM probably is cheaper to execute than an RFM that transforms a piece of code, which is modified with public inside a deep inheritance hierarchy. The refactoring tool then could try to arrange cheap RFMs first in the sequence of RFMs because this can help that for a longer time, memory is not exceeded by loading code – this can reduce the number of buffer misses and increase performance.

6.2.2. Case Studies

We prototypically implemented the basic concept of the algebraic optimization approach (e.g., the prototype so far does not check that the sequence to optimize succeeds). The prototype currently optimizes the RFM sequence of an SPL product separately before the product is generated (for details, see [KLS10b, Lia10]). We report now on the evaluation of the prototype in a number of case studies.

[13]Unmodified members can be accessed only from code of the same package or code of subclasses of respective host classes (cf. Sec. 2.1, p. 8).

Study Setup

We studied SPLs of different sizes and purposes. For each SPL, we performed three basic steps: First, we measured the time which the composer tool needed to generate an SPL product with an unoptimized sequence of feature modules. Second, we measured the time which the optimizer prototype needed to optimize the feature-module sequence. Finally, we measured the time which the same composer tool needed to generate the SPL product with the optimized sequence of feature modules. We show the measured runtimes in Table 6.3.[14] For each of our studies, we selected only one solitary Jak-like feature module because the optimizer does neither alter Jak-like feature modules nor their order.

We evaluated our concept and prototype in four case studies: To prove the concept, we studied the generation of a conceptual list implementation without functionality. To analyze whether SPLs with functionality can be optimized with our concepts, we studied a product of an SPL of arcade games which has small-scale Jak-like feature modules. To analyze whether large-scale Jak-like feature modules and long RFM sequences can be optimized with our concepts, we studied three products of an SPL version of a large-scale Eclipse module; a version with a number of RFMs. To show that optimizing feature-module sequences can impair the generation time of programs, we report on a product of the SPL version of the archive-access module ZipMe.

Proof of concept. We analyzed three products of an SPL of conceptual list implementations (did not define functionality); in these products, different RFM sequences transformed the code that was generated by the solitary Jak-like feature module. In a first SPL product, the optimizer fused an RFM, which extracted the API artifact AbstractList from a class List, with multiple reordered RFMs, which all renamed the extracted API artifact. In the optimized feature-module sequence, a new RFM extracts the API artifact with the final name in the first place. In a second SPL product, the optimizer could not fuse two RFMs because predecessor dependencies prohibited it to reorder the RFMs.

Functionality in Jak-like feature modules. We analyzed a product of an SPL of arcade games for personal computers and cell phones: TankWar. The study is small-scale but includes functionality (in contrast to the proof-of-concept study). The concepts worked well for this study such that we could reduce the time a tool needs to generate the analyzed SPL product.

[14]We performed the measurements on a computer with an Intel Core 2 CPU T5500 with 1.66GHz and 0.99GB RAM running a Microsoft Windows XP SP2. The measurements of Tab. 6.3 are *averages* over 10 runs, listed one by one elsewhere [Lia10]. The optimizer generates optimized and unchanged RFMs into a temporary directory but does not copy the Jak-like feature modules; we copied the Jak-like feature modules *manually* into the directory to measure the time the tool needs to execute the optimized feature-module sequence (to generate the SPL product). We did not implement the detection of name capture so far. We implemented five fusion rules so far.

Table 6.3.: *Tool runtimes for feature-module sequences (in* ms*) [from KLS10a].*

Program	**#SLOC**$^{\alpha}$	**#RFMs (unopt.)**	**#RFMs (opt.)**	**Composer runtime (unopt.)**	**Composer runtime (opt.)**	**Optimizer runtime**	**Pure optimization runtime**
Conceptual List (a)	19	5	2	12018.6	9870.4	8934.6	9.4
Conceptual List (b)	19	8	4	12840.7	9546.8	9401.4	9.5
Conceptual List (c)	19	10	4	16359.3	10412.3	9074.6	20.3
TankWar	∼1K	10	4	31934.2	14093.6	8206.3	18.8
Workbench.texteditor (a)	∼16K	10	4	172162.4	83561.1	18749.9	17.2
Workbench.texteditor (b)	∼16K	17	3	253831.2	59731.2	18448.4	23.3
Workbench.texteditor (c)	∼16K	55	3	769617.5	61292.1	77632.7	101.4
ZipMe	∼3K	3	3	20461	20281.4	7867.1	10.8

$^{\alpha}$lines of source code without RFMs

Large-scale Jak-like feature modules. We analyzed three products of an SPL version of the Eclipse module 'workbench.texteditor'; a version which includes a large-scale Jak-like feature module. While generating the SPL products with the unoptimized RFM sequences, the composer tool executed 10 to 55 RFMs. The optimizer prototype reduced the number of RFMs in all analyzed SPL products to three; optimization did reduce the time a tool needed to generate the products.

Impairing optimization. We analyzed a product of an SPL version of the archive-access module ZipMe. In this study, the optimizer could not fuse any RFMs. As a result, the optimizer runtime must be added to the time the composer tool needs to execute the unoptimized feature-module sequence; optimization thus is derogatory in this case.

Discussion

In Table 6.3, we list the runtimes that we measured for the optimizer prototype and relate them to the runtimes that we measured for the composer tool executing (a) the unoptimized and (b) the optimized feature-module sequences. In some studied cases, runtime decreased with optimization by up to 81 % (in the 'workbench.texteditor'(c) case) and thus our optimization was beneficial here.[15] In other cases, runtime increased with optimization by up to 56 % (in the conceptual-list(a) case).[16] However,

[15]81 % = (1-(61292.1ms+77632.7ms)/769617.5ms)
[16](-56 %) = (1-(9870.4ms+8934.6ms)/12018.6ms)

our optimization approach did not fail. The runtime increased mainly because the optimizer prototype executes separately before the composer tool. For that, times to load RFMs in the optimizer prototype and times to store the result of the optimizer prototype contribute to the measured tool runtimes though these times vanish once both tools are integrated (possible future work).[17]

To emulate the case, when the optimizer prototype is integrated with the composer tool, we measured the time which the optimizer prototype needed to only optimize the loaded RFM sequences (cf. Tab. 6.3, column *Pure optimization runtime*). As a result in all but the ZipMe study, runtime decreased remarkably with optimization; for the 'workbench.texteditor'(c) case, the runtime decreased by even 92 %.[18]

Our measurements suggest that the optimization benefit corresponds to the size of Jak-like feature modules; the reason is that the highest runtime reductions could be achieved for the SPL products which have the Jak-like feature modules of the largest scale ('workbench.texteditor'(a)-(c)). We assume that the refactoring tool needs more time to alter and traverse a lot of code than to alter and traverse less code, and that saving this time has a higher influence with large-scale RFMs.

Our measurements suggest that the optimization benefit corresponds to the length of RFM sequences; the reason is that the highest runtime reductions could be achieved for the SPL products for which the longest RFM sequences must be executed ('workbench.texteditor'(a)-(c)). We assume that a growing number of RFMs increases the potential for fusible RFMs.

Threats to validity. Our measurements depend on how much time the composer tool needs to load RFMs; if loading an RFM takes a lot of time, then reducing the number of RFMs saves a lot of time. Our measurements depend on how much time the composer tool needs to execute RFMs; if to execute a single RFM takes a lot of time, then reducing the number of RFMs saves a lot of time. To execute RFMs, we used the only RFM composer tool we know of (cf. Sec. 4.2.3); this tool was written for flexibility and not for performance and so other composer tools may behave differently.

The studies were just in part implemented independently from this evaluation (i.e., many studies existed before but we added RFMs to them especially for this study). Evaluating sequences of others remains future work. Nevertheless, we were able to show that optimizing sequences of feature modules is possible and can be beneficial.

The removal of RFMs includes the removal of precondition checks (if implemented). We did not distinguish execution times for verification and transformation so far.

[17] Currently, the optimizer prototype *and* the composer tool load the RFMs; an integrated tool would load the RFMs only once and pass them in memory. Currently, the optimizer prototype writes the optimization result to hard disk; an integrated tool would pass the result in memory.

[18] 92 % = (1-(61292.1ms+101.4ms)/769617.5ms)

6.2.3. Summary

SPL users can trigger sequences of feature modules by selecting features. The sequences a user can trigger may take more time to execute than necessary. In this section, we showed how a tool can shorten the user-triggered sequences of RFMs in order to reduce the time a composer tool needs to execute them. We implemented a prototype and evaluated this prototype in a number of case studies. We observed that optimization reduced program-generation time in many cases though not all.

6.3. Multi-Language Support for RFMs

Section 6.3 is based on the Diploma thesis of Hagen Schink [Sch10], which was conducted and supervised in the course of this thesis; extended versions have been published elsewhere [SK10, SKSL11].

In the previous chapters and sections, we analyzed RFMs for SPLs that were written using one programming language, only. That is, we concentrated on SPLs written in the Jak programming language (note that the concepts presented so far work for every language for which we can describe refactorings in modules). However, a single program may include (a) pieces of code written in different programming languages of one programming paradigm, such as C++ (OOP language [Str91]) and Java (OOP language [GJSB05]), (b) pieces of code written in languages of different programming paradigms, such as Haskell (functional programming language [Jon03]) and Prolog (logic programming language [CR96]), and (c) noncode artifacts, such as documentation and models [BSR04, CJ08, For08, GBP04, KWDE98, SKL06]. We call a program written in more than one language *multi-language program*. Common examples of multi-language programs include Java programs that call methods written in C++ [Lia99], C++ and Java programs that execute database queries written in the *structured query language (SQL)* [Ora05, Sun06], and Java programs that manipulate artifacts written in extensible markup languages [BEA03, CJ08].

In a multi-language program, artifacts written in different languages can interact (e.g., due to method calls or due to overloaded mechanism meanings). Refactorings exist for presumably all languages that support structured artifacts [LBL06, MT04]. When tools implement these refactorings and transform only the artifacts of one language, relations between different artifacts of a multi-language program may break and functionality may change (i.e., the refactorings would fail). For example, when a refactoring tool renames a method in Java artifacts but does not update the C++ artifact, which calls the Java method, then this C++ artifact has a dangling reference after the refactoring, and linking errors or runtime errors can occur. As a result, we and others [BSR04, TBD06] conclude: To generate a multi-language program from feature modules, feature modules must generate and alter *all* documents of that multi-language program consistently.

As a first step, we studied whether RFMs can be used to generate a multi-language program and interestingly, we observed several problems: The refactoring of multi-language programs (we call such refactorings *multi-language refactorings (MLRs)*) may involve to alter artifacts which control cross-language interactions in a multi-language program (i.e., artifacts that do neither define program functionality nor program state). MLR may depend on the existence of artifacts written in a certain language rather than on the existence of certain artifacts in a language. MLR may depend on correct predictions on runtime behavior of methods.

Based on our analysis, we prototypically extended our composer tool such that RFMs can transform certain multi-language programs; we evaluated the prototype in case studies and discuss its limitations. We conclude that tools can only automate general MLR with programmer interaction. Refactorings for specific combinations of languages are still possible to automate completely.

6.3.1. Analyzing MLR for a Hibernate Application

We extended a product of the GPL to be a multi-language program.[19] Specifically, we extended the product to include artifacts written in the OOP language Java, the functional programming language Clojure[20], and the declarative database-query language SQL [CB74]. We then refactored the extended GPL product *manually* to identify challenges of automating MLR.

In this study, we used the module Hibernate[21] to integrate Java artifacts with the database and thus with SQL artifacts. That is, we used SQL artifacts to first create database tuples; we then used Hibernate to create objects in the Java program from the tuples (Hibernate also could update the tuples on behalf of changes to the Java objects but we did not use this facility). We used Hibernate in property-access mode [KS06]; that is, we annotated classes to define that the properties of these classes are to be stored in the database, and we defined access methods to specify and access these properties. As an example, in Figure 6.9a, we show, how we annotated class Vertex to make Hibernate synchronize Vertex objects with tuples in the database table vertices (Lines 1-2); without the @Table annotation, Hibernate would try to synchronize Vertex objects with tuples of a table Vertex. In Figure 6.9a, we also show the access methods Vertex.getName()::String and Vertex.setName(String)::void, which together define the property *name* of Vertex objects (by name correspondence) – Hibernate stores the property *name* of Vertex objects in column name of the database table vertices. We refactored the multi-language program starting from the Java and starting from the SQL artifacts; we now report on the three most interesting cases (more cases are discussed elsewhere [Sch10, SK10, SKSL11]).

- We applied a refactoring to Java artifacts; specifically, we renamed the method Vertex.getName()::String into getVertexName to distinguish this method from

[19] We did not actually extend a GPL product. Instead, we implemented a sample multi-language program HRManager from scratch [Sch10, SK10, SKSL11], similarly to the example in [KS06]. We transfer the results from HRManager to the GPL example to keep the examples consistent in this thesis. That is, for this thesis we map names of HRManager to names of the GPL.

[20] `http://clojure.org/` (accessed: November 26,2010)

[21] `http://www.hibernate.org/` (accessed: November 26,2010)

```
 1 @Entity
 2 @Table(name="vertices")
 3 public class Vertex implements Serializable {
 4  private String name;
 5  public void setName(String name) {
 6   this.name = name;
 7  }
 8  public String getName() {
 9   return name;
10  } }
```

(a) Class of objects to store.

```
1 CREATE TABLE vertices(
2  name varchar(255)
3 );
```

(b) Database table to store Vertex *objects.*

```
1 CREATE TABLE vertices(
2  name varchar(255) CONSTRAINT std_name DEFAULT
     'noname'
3 );
```

(c) Database table to store Vertex *objects after Introduce-Default-Value refactoring.*

Figure 6.9.: *Hibernate annotations to persistently store properties of objects of class* Vertex *in the database table* vertices *[adapted from Sch10].*

other methods. We had to update a piece of Clojure code which referenced Vertex.getName()::String; however, as variables in Clojure do not need to have a type [VS10], we *had to know* that the Clojure code actually references the Vertex method and only this method – a tool, however, cannot predict this safely at development time [RBJ97, TDDN00].

We had to refactor the complete multi-language program a *second* time to rename method Vertex.setName(String)::void into setVertexName; this was necessary because Hibernate otherwise would alert that there exists a property *VertexName* with a get-access method but no set-access method. Hibernate requires such method *pair* in property-access mode [KS06]; if Hibernate would have been used in other modes than property-access mode, different actions would be required from a refactoring tool.

Finally, we had two options to maintain the consistency between the Java artifacts and the SQL artifacts: We could (a) rename the column name in database table vertices to VertexName and update a number of SQL statements[22] or we could (b) add a Hibernate-specific @Column annotation to method getVertexName to instruct Hibernate to synchronize the property *VertexName* with the database column name. In this study, we added the annotation.

- We applied a refactoring to SQL artifacts (Fig. 6.9b); specifically, we defined that the default value for the column name of a tuple of the table vertices should be noname (Fig. 6.9c, Line 2; Introduce-Default-Value refactoring [Amb03]). To maintain consistency between artifacts of the multi-language program, we had to initialize the field name of class Vertex with the new default value noname; however in doing this, we could not guarantee to maintain the functional-

[22]As object-oriented refactorings commonly do not discuss runtime entities derived from classes (objects), we do not discuss runtime entities of database schemas (tuples).

```
String delim = askUser();
public void setVertexName(String edgeID) {
 if (name.indexOf(delim) != −1) {
  this.name = edgeID.substring(0,edgeID.indexOf(delim));
  this.neighborName = edgeID.substring(edgeID.indexOf(delim),edgeID.length());
 } else {
  this.name = edgeID;
 } }
```

Figure 6.10.: *Method for which the result is difficult to predict automatically.*

```
select sum(i.quantity) from Inventory i
```

(a) Original statement.

```
select sum(i.numberOfProducts) from Inventory i
```

(b) Transformed statement.

Figure 6.11.: *Manual adjustment of an MLR-study artifact that was written in the Hibernate query language [from SK10].*

ity of those Java methods that behave in a certain way when the field Vertex.name::String is uninitialized (value is null); these methods must change to make them work with noname as new default value but this might be difficult (e.g., when noname is used as a value already in those methods though not as default). Updating a multi-language program automatically, which includes such methods, might be impossible.

The Java member variables to initialize and their new initialization values cannot be predicted, in general. That is, the fields to set may be calculated at runtime by the multi-language program's methods and the results of these methods are difficult to predict (static analyses of programs are limited [LSH98]). For example in Figure 6.10, we show a possible set-access method setVertexName that calculates the fields to set and that transforms its parameters; it is difficult to predict for this method which fields should be set by the refactoring and how.

- We applied a refactoring to the SQL artifacts; specifically, we removed a table that was no longer needed (Remove-Table refactoring [Amb03]). To maintain consistency, we had to remove the class which corresponded to the removed table according to the Hibernate configuration file. In addition, we had to update this tool-specific (Hibernate-specific) configuration file to avoid runtime errors.

6.3.2. Case Studies

We prototypically extended the implementation of some refactoring types of our composer tool to support multi-language programs (according RFMs now implement

Table 6.4.: *Data on programs used to evaluate RFMs for MLR [adapted from SK10].*

Program	**#Java-SLOC**$^{\alpha}$	**Refactorings (Start language)**
HRManager	428	Remove Table (SQL), Rename Method (Java)
Seam-gen project	164	Rename Method (Java)
Seam DVD store	1714	Push Down Method (Java), Rename Method (Java)
Seam space	1367	Rename Method (Java)

$^{\alpha}$lines of source code

MLRs to some extent as we explain next). We used this prototype to execute RFMs on the multi-language program, which we just studied (i.e., HRManager [SKSL11]), and on different sample multi-language programs of the application framework JBoss Seam[23]. All multi-language programs included at least Java artifacts and Hibernate-related artifacts (Java annotations). Due to technical limitations of our prototype, we made minor adjustments manually (e.g., we adjusted an artifact of the Hibernate query language shown in Figure 6.11a to become the artifact shown in Figure 6.11b, and we adjusted SQL queries that occurred together in a file). We summarize our studies in Table 6.4.[24] For simplicity, our prototype applies Hibernate-specific Java annotations to avoid updating a number of SQL artifacts.

6.3.3. Summary

We analyzed how a refactoring of an RFM can transform programs that are written in multiple languages. We found that refactorings, which execute on artifacts written in one language, may impose functionality changes for artifacts written in other languages. We observed that updating artifacts on behalf of a refactoring might require predictions on method results – these predictions are difficult to make automatically. We observed that tools, which implement MLRs, must maintain configuration files of modules, which manage the interaction between artifacts in the refactored code; these tools further must act according to the configuration of these modules in order to carry out a refactoring correctly.

In our analysis, we observed a trade-off between supported refactorings and supported languages: Either refactoring tools can support a number of refactorings, then these tools in our experience can cover only a limited number of languages; or refactoring tools can cover a high number of languages, then these tools can only cover a few refactorings; to support a high number of refactorings for a high number of languages, the tool must consider a high number of concepts for every of these languages – this hardly scales in terms of implementation effort. As a result, also the extension that we prototypically implemented for the RFM composer tool is specific for the analyzed multi-language programs; for example, it expects Hibernate to

[23]`http://seamframework.org/` (accessed: November 26,2010)

[24]In parts, the studied multi-language programs include artifacts written in languages our prototype does not support so far. We updated these artifacts manually. In these cases, our prototype *reduced* the manual effort to carry out the MLR but did not carry out the MLR automatically completely.

be used, it expects Hibernate to be used in property-access mode, and it expects Hibernate annotations to be available.

6.4. Summary

The first challenge we discussed and tackled in this chapter was the detection of incorrect code in SPLs and SPL products, and its effective correction; we found that starting to detect and correct errors at the level of a single product can be beneficial when the SPL involves only Jak-like feature modules and RFMs. We showed how programmers can semi-automatically correct the SPL based on corrections made in a single product.

The second challenge we discussed and tackled was the reduction of the time that a tool needs to generate products of an SPL from a sequence of Jak-like feature modules and RFMs; we reordered and fused RFMs and we defined what should be analyzed in Jak-like feature modules in order to reduce the time a tool needs to generate products of such SPL.

The final challenge we discussed was the consistent configuration of programs with RFMs when the programs include artifacts written in different languages; we observed that refactoring tools might need predictions on algorithm results and are limited by a trade-off between supported refactorings and supported languages. We were able to provide a partial solution for MLR of multi-language programs. To gain a fully fletched solution, further research is required.

7. Related Work

We now review work that is related to the ideas and concepts we discussed in this thesis. We group the work with respect to the discussions of the prior chapters.

7.1. Related Work on RFM Concept

In Chapter 4, we proposed to integrate refactorings with feature modules of SPLs in order to configure the structure of SPL products (i.e., we described refactorings in SPL modules and used them to generate modules/programs). We reviewed related approaches on module integration and showed their weaknesses in Section 3.3. RFMs address these weaknesses (cf. Chap. 4 & 5). That is, SPLs with Jak-like feature modules and RFMs support an integrated, feature-driven configuration of a module's functionality and structure.[1] We now distinguish the RFM concept from the approaches partly discussed in Section 3.3 beyond the strengths and weaknesses which we already discussed.

Wrappers. Wrappers can make a module compatible with an environment when this environment expects the module to be written in a different language, whereas RFMs cannot make a module compatible with an environment when this environment expects the module to be written in a different language. Wrappers can remove incompatibilities which RFMs cannot remove; for example, wrappers can be independent classes which provide the functionality of one class of a module, whereas RFMs cannot generate *independent* classes which provide the functionality of one class of a module. Wrappers can add code of missing functionality to programs [BCL10, ESWH04], whereas RFMs cannot – however, RFMs are integrated with Jak-like feature modules and these Jak-like feature modules can add such code. Wrappers cannot remove incompatibilities which RFMs can remove; for example, wrappers cannot provide a new name for a method without introducing a new scoped name for the method's hosting class simultaneously. Wrappers cannot provide access to code of modules that is not part of the module's API (e.g., to blocks inside methods), whereas RFMs can provide access to such code by extracting it into new methods. Wrappers commonly cannot guarantee to maintain functionality of a

[1]Integrated, feature-driven configuration (cf. Sec.3.3): Users should be able to configure the functionality and the structure of a module. Thereby, users should be able to define configurations without knowing anything about the module's implementation. Thus, the configuration of the structure and the functionality of a module should be based on the concept of features. Finally, such an approach should ensure that decisions regarding the functionality of a module do not affect decisions regarding the structure of this module, and vice versa.

module, whereas RFMs do. Wrappers are commonly developed by programmers of the environment, whereas RFMs are intended to be developed by programmers of the reusable module; programmers of RFMs thus need to know the module environments. Thereby, programmers of modules *are pointed* to the pieces of code inside the environment, which their modules should interact with, by compiler errors on dangling references; programmers of environments or wrappers are *not pointed* to pieces of code inside the module their environment should interact with.

Automated extension or redefinition of APIs. Researchers explored how to extend APIs of modules (e.g., to allow environments to access private members of the modules) [Her08]. Technically, researchers can define pieces of code in the reused module that should be extended and can define extensions to these pieces [HM07]. In these approaches, the old APIs of modules remain valid after integration, whereas with RFMs, old APIs do not remain valid.

Researchers defined meta-programs that transform names, parameter values, and return values of methods of a module as well as names and values of fields; as a result, such module can be manipulated by a generic algorithm [Nov95, Nov97]. Programmers must verify *on their own* that their meta-programs maintain functionality (if intended); for example, programmers must verify *on their own* that parameter values and return values are transformed consistently. After the transformation, only the new names remain valid. The meta-programs of these approaches have not been used to implement configurable features of SPLs, whereas RFMs have been used this way. With RFMs, programmers do not define meta-programs but only parameters for meta-programs (of refactoring types). After the transformation of RFMs, only the new names remain valid.

Refactoring for module integration. Programmers can record a sequence of refactorings in a refactoring history. The refactoring history then groups refactorings by programs and execution time. Programmers can select refactorings in a refactoring history for a *refactoring script* [Wid07].[2] Refactoring scripts finally can update legacy copies of the refactored program or legacy environments of a refactored module [Dev09]. In contrast to refactoring histories, a feature model groups refactorings (of RFMs) by the features they implement. The transformations (refactorings) in a refactoring history or refactoring script do not allow programmers to configure modules with respect to functionality.

Researchers execute refactoring-like transformations on a program to derive a representation for this program which is at a higher or lower level of abstraction (e.g., they derive COBOL II code from Assembler code or derive code from specifications) [PR01, PR03, WB95]. In these approaches, refactorings and transformations

[2]Eclipse documentation (accessed: January 31,2011): `http://help.eclipse.org/help33/-topic/org.eclipse.jdt.doc.user/tasks/tasks-240.htm`, `http://help.eclipse.org/help-33/topic/org.eclipse.jdt.doc.user/tasks/tasks-241.htm`, `http://help.eclipse.org/-help33/topic/org.eclipse.jdt.doc.user/tasks/tasks-242.htm`

are not related to selectable features of an SPL; that is, in these approaches, users know the code which they change as well as the refactorings.

Annotative SPL approaches. Researchers implemented SPLs using annotative approaches.[3] Annotative approaches implement enumerative program transformations because by analyzing the code, which an annotation embraces, one knows all the pieces of code this annotation transforms. Annotative approaches implement monotonic program transformations because they only involve code removal; thus, they do not support features of refactorings well. To implement an RFM "Rename class Vertex into ADT" in an SPL implemented with an annotative approach, the program with all features must include a class Vertex *and* a class ADT, or a class with two alternative names; further, the SPL code must include for every reference to Vertex an alternative reference to ADT. This either introduces code clones or fine-grained annotations (described to be complex [KAK08]).

Module migration beyond structure. Researchers performed semi-automated visual analyses to migrate user-interface modules [Ger09]. The migration concerns the positions of user-interface elements, such as buttons, and their replacement. Instead, RFMs change the structure of the code of a module.

Aspect refinement vs. RFM refinement. Researchers refined program transformations (aspects) of aspect-oriented programming, before these researchers executed the synthesized transformations to generate SPL products [AKLS07]. Similarly, we refine program transformations (refactorings) of SPLs, before we execute the synthesized transformations to generate SPL products. The (synthesized) aspects may generate code with new functionality within their input programs, whereas (synthesized) RFMs do not – RFMs only restructure code.

7.2. Related Work on Algebraic-Property Analysis

In Section 4.2, we analyzed refactorings for SPLs and SPL products with respect to algebraic properties. To be precise, we defined terms and operations of refactorings and Jak-like feature modules. Based on these terms and operations, we analyzed whether there are identity operations in the domain of refactorings, whether there are inverse operations in the domain of refactorings, and whether there are refactorings that distribute over Jak-like-feature-module execution.

Algebraic properties of program transformations. There are a number of desired algebraic properties for refactorings with respect to Jak-like feature modules of SPLs (e.g., distributivity) [BAS08, Bat07b, Bat07c, Bat09, BS07]. Researchers analyzed

[3]Common annotative approach to implement SPLs (cf. Sec. 2.2.2, p. 12): (a) Implement a program with all features of the SPL; (b) annotate code with respect to features; (c) remove the code of unselected features when generating a product.

that *rewrite* transformations do not distribute over Jak-like-feature-module execution [BS07]. Researchers formally prove algebraic properties of *Jak-like* feature modules [ALMK10]. Our examples and formal proofs show that programmers cannot distribute refactorings over Jak-like-feature-module execution in general; we were even able to show that, in theory, distributivity holds only in exceptional cases. Beside distributivity, we proved the existence of identity operations and inverse operations in the domain of refactorings.

The operations of refinements and transformations have been analyzed to distribute in several cases [TBD07]. For example, researchers were able to generate equal controllers for their domain-specific language of portlets in both ways: (a) execute Jak-like feature modules to generate a state chart from state-chart pieces and translate this state chart to the language of the controller, or (b) translate the state-chart pieces in the Jak-like feature modules to the language of the controller and execute the translated Jak-like feature modules. That is, researchers showed among other things that language translation can distribute over Jak-like-feature-module execution. We proved that refactoring operations do not generally distribute over Jak-like-feature-module execution; specifically, we were able to show that, in theory, refactorings distribute *rarely* over Jak-like-feature-module execution.

Researchers describe an algebra on patches that allows them to merge *concurrent* patch sequences; the resulting integrated sequence can be executed on a program afterwards [Lyn06]. In contrast to patches, refactorings in RFMs have sophisticated preconditions that must stop the refactoring of the RFM if they are not met in the input program of this RFM; the properties which researchers showed for patches thus cannot be assigned to refactorings smoothly. In contrast to this related work, we analyzed algebraic properties of refactorings with respect to Jak-like-feature-module execution. **With respect to our discussion on correcting functionality errors of SPL products (cf. Sec. 6.1):** The algebra on patches allows programmers to invert transformations in concurrent sequences of transformations, but we invert transformations (refactorings and refinements) in a single sequence. The algebra on patches allows programmers to integrate patches with a stand-alone program, whereas we integrate corrections with all products of an SPL (i.e., with up to millions of programs). The algebra on patches allows programmers to reorder and invert transformations to generate an integrated sequence of program transformations; we reorder and invert in order to propagate corrections from products of an SPL to feature modules of this SPL. **With respect to our discussion on reducing the generation time of SPL products (cf. Sec. 6.2):** The algebra on patches allows programmers to reorder and fuse program transformations of concurrent sequences, but our optimization approach reorders and fuses transformations in only one transformation sequence. The algebra on patches allows programmers to integrate different sequences, whereas we aim at reducing the length of a single sequence.

Researchers describe an algebra of model composition and define algebraic properties for model-composition operators [HKR$^+$07]. In contrast, we define an algebra on refactorings and code composition of Jak-like feature modules.

Refactoring of SPLs. Researchers refactor SPLs at the level of models as well as at the level of code. At the level of models, researchers restructure feature models [AGM+06]; for example, they remove features which cannot be selected for any legal product. At the level of code, researchers execute feature-oriented refactorings to transform a stand-alone program into a set of Jak-like feature modules, or to restructure Jak-like feature modules (cf. Sec. 2.3, p. 16). Aspect-oriented refactoring transforms a program into a base program plus aspects [CBS+07, GJ05, MF04, MF05], or aspect-oriented refactoring restructures aspects. RFMs apply at code level. RFMs, however, do neither separate programs into Jak-like feature modules nor into aspects. RFMs implement *object-oriented* refactoring types such as Rename Method. None of the above studies discussed problems which we proved formally for refactoring Jak-like feature modules. **With respect to our discussion on the RFM concept (cf. Chap. 4):** RFMs do not transform the feature modules of an SPL but only their execution results (individual products of SPLs).

Researchers rename pieces of code in an SPL that is implemented with an annotative approach [Vit03]. These researchers faced problems (e.g., they faced situations in which they could not decide which pieces of code to rename). To overcome the problems, they generate parts of different SPL products and in the worst case they generate a number of products (which can count exponential in the number of features). We formalized refactorings and Jak-like-feature-module execution in an algebra to find general underlying concepts and challenges for their combination (also underlying the problems these researchers faced for renaming).

7.3. Related Work on NFP Configuration

In Section 4.3.2, we discussed how RFMs can be used to configure SPL products with respect to NFPs. We demonstrated that RFMs can be used to configure SPL products with respect to the quality of their code, with respect to their performance and footprint. In the course of this study, we generated RFMs in order to make a difference for NFPs that is noticeable.

Determining NFPs. Researchers found that NFPs of programs are difficult to determine [SRK+08] and that NFPs are even more difficult to predict for SPL products [SSS07]. We do not predict values for code with respect to an NFP metric (as others did before [SRK+08]) but proposed heuristics for how to alter code such that it performs better with respect to NFP metrics; we alter code with refactorings.

Researchers optimize programs with respect to NFPs after they attributed the features with values for NFPs [BSTR07, BTR05, ZJ05, ZJY03]. Based on these values, the researchers calculate feature selections of which the products satisfy NFP constraints best. In addition, researchers use these feature-attribute values to prune the set of selectable features during program configuration [WDS09, WSWN07]. Most above approaches predict values for products with respect to NFP metrics. We configure NFPs of SPL products without attributing features. We also do not predict values for generated programs with respect to a given NFP metric. We do not prune the set of SPL products during program configuration. Nevertheless, we improve code with respect to NFPs.

Researchers restructure sets of SPL products such that these products expose smaller footprints. Specifically, when multiple products of an SPL are used simultaneously in an environment, the researchers merge the common parts of these products in a module [LB04]. In contrast, we target NFPs beyond footprint and we improve individual SPL products with respect to NFPs.

Optimizing stand-alone programs. Researchers optimize stand-alone programs with respect to an NFP by selecting transformations, which alter the programs with respect to this NFP; for example, researchers select transformations which perform partial evaluation [Smi90, Smi91, WS97]. We analyzed how users can configure NFPs at the level of features (i.e., users do neither need to know any code nor any refactoring/transformation).

Researchers optimize high-level descriptions of (configurable) programs with respect to NFPs [CDCv03, KPS08, RPB09]. In contrast to their work, we optimize code. In contrast to work on compiler optimizations [DC94, Sch73, Sri92, WLQ09, WS91] and platform-driven code generation [PMJ$^+$05], we analyzed the configuration of NFPs beyond footprint and performance (e.g., we configured code with respect to its quality).

Removing style problems in stand-alone programs. Researchers propose refactorings [NSCP06] and generate sequences of refactorings [Pér08] to reduce the number of occurrences of well-known style problems in stand-alone programs. We discussed the generation of RFMs too but not to configure code quality (includes code style). To configure code with respect to quality metrics, we proposed to define refactorings (in RFMs) manually because generating these refactorings is controversial [Opd92, RBJ97]. In contrast to the above work, we *configure* modules with respect to NFPs and discussed NFPs beyond code style.

Alternative implementations. Researchers configure code with respect to NFPs by selecting one out of many alternative implementations for the same required piece of functionality [SRK$^+$08, SSS07, ZJY03]. The researchers then maintain all alternative implementations in parallel which we argue is laborious and error-prone. RFMs *transform* a program towards a different structure so programmers must implement and maintain a piece of functionality only once in Jak-like feature modules.

7.4. Related Work on Safe Composition

In Section 5.3, we discussed concepts that tools could implement to verify that every product described in a feature model can be generated without error from feature modules and can be compiled. Accordingly, we analyzed Jak-like feature modules and

RFMs and recorded in decision trees the effects of executing the modules on scoped names of pieces of code. We derive composition constraints for feature modules from these trees and verify these constraints with respect to the feature model using a SAT solver.

SPL tests. Researchers check in one step that all products of SPLs can be generated without error and can be compiled. These researchers focus on SPLs in which program transformations monotonically add code (in Jak-like feature modules) [AKGL10, DCB09, TBKC07] or monotonically remove code (in annotative SPL approaches) [CP06, KA08, KKB08], and they focus on program transformations that enumerate all the pieces of added or removed code. We follow the same goal but focus on SPLs in which program transformations nonmonotonically add *and* remove code, and in which program transformations do not always enumerate all the pieces of code they transform.

In feedback-driven testing [MPY$^+$04], researchers test SPLs incompletely but in a structured process. Feedback-driven testing guides a programmer in the case when an SPL product is found in error: The programmer is guided to test and correct every product that is in error and all products that differ from a product in error in exactly one feature; this becomes a recursive process which stops as soon as all *tested* products are free of error. With feedback-driven testing, there is neither a guarantee that the identified error has been corrected in every SPL product nor that every SPL product is correct. Feedback-driven testing can help to detect compilation errors *and* errors in functionality. In contrast, we investigated in safe composition. Safe composition is an exhaustive search and verifies whether *every* product of an SPL can be generated without error and can be compiled. In contrast to feedback-driven testing, safe composition cannot detect errors in functionality in any product. Further, our safe-composition approach does not generate any product of an SPL.

In incremental pairwise testing [OMR10], researchers test SPLs incompletely but in a structured process. According to incremental pairwise testing, programmers generate and test some SPL products such that every pair of features is tested together in at least one product. Incremental pairwise testing, thus, finds errors when (a) two features together act erroneously in a tested SPL product, when (b) two features together always act erroneously, or when (c) one feature on its own acts erroneously. When two features together act erroneously but only in a product which was not tested, this error goes unnoticed. In contrast, we investigated in safe composition. Safe composition is an exhaustive search and verifies whether *every* product of an SPL can be generated without error and can be compiled. In contrast to incremental pairwise testing, safe composition cannot detect errors in functionality in any product. Further, our safe-composition approach does not generate any product of an SPL.

Researchers verify that every piece of code embraced in #ifdef directives can be selected in a feature model for at least one SPL product [TSSPL09]. In contrast to our work, these researchers do not focus on the configured code and so their work is not on compilability of products.

Researchers describe manually dependencies between modules using API definition languages (cf. Sec. 2.1, p. 6), design rules [Bat05, BG97], and contracts [Mey92]; tools can use these descriptions to prevent the selection of incorrect combinations of modules at program-generation time [Bat05, HKA10, vdS04, WSWN07]. A dependency described manually can describe that a module depends on unknown modules (which have particular properties). A dependency described manually can describe compilability constraints and constraints in functionality. Our safe-composition approach detects dependencies between feature modules automatically. The detected dependencies are only on whether SPL products can be generated without error and can be compiled; the dependencies do not guarantee that products provide a certain piece of functionality. A dependency, which a tool can detect with safe-composition techniques, can describe only that a module depends on other *known* modules. Manual descriptions can be used to detect errors first at program-generation time, whereas our safe-composition approach can detect errors already at the time the SPL is developed.

Program tests and program verification. Researchers analyze properties of individual program transformations [LJWF02] and of stand-alone programs [EM04, SdML04] with model-checking techniques, and further analyze properties of stand-alone programs with static-analysis techniques [APH$^+$08, Eff11, EM04, LSH98]. In contrast to their work, we verify a high number of programs (all products of an SPL) simultaneously. Our analysis, however, is only on compilability. We used SAT-solver technology.

Researchers formalized individual refactorings and preconditions of individual refactorings in order to prove that these refactorings do not alter functionality [Li06, MEDJ05]. Researchers verify with a functionality model, whether transformations change the functionality of a stand-alone program [RCD09]. We rely on refactorings to not alter functionality and verify that they can execute without error inside a flexible sequence of refactorings (i.e., inside a sequence in which refactorings might be selected but need not be selected). We verify with preconditions whether refactorings change functionality in any product of an SPL.

Researchers apply unit tests to evaluate the functionality of programs. Unit tests are samples and so programs still can fail when a unit test succeeds [TTS$^+$08]. Researchers decompose tests with respect to features and synthesize them [OG09]. These approaches test individual products of an SPL one by one, but they so far do not test all products of an SPL in one step. Our safe-composition approach verifies all products of an SPL in one step. Our safe-composition approach does not evaluate functionality but only compilability.

Researchers apply program-verification techniques to evaluate the functionality of programs. Program-verification techniques commonly require programmers to manually annotate their code with constraints; after this, tools can proof consistency between a piece of code and its annotation. Researchers can reuse proof steps for different SPL products [BKS10]. Recently, researchers investigated even the decomposition of proofs [TSKA11]. While these approaches verify stand-alone programs, they so far do not verify all products of an SPL in one step. Our safe-composition approach verifies all products of an SPL in one step. Our safe-composition approach does not evaluate functionality but only compilability. Those approaches further are not on nonmonotonic, nonenumerative feature modules.

Nonmonotonic feature modules. Delta modules are nonmonotonic program transformations used to implement SPLs [SBB$^+$10, SD10]. Delta modules can generate classes, remove classes, and change classes; to change a class, delta modules can alter the superclass declaration and generate, remove, and rename members of the class (renaming does not update references so it is not a refactoring [SBB$^+$10]); delta modules also can synthesize models [HRRS11]. Delta modules enumerate all the pieces of code they transform, whereas RFMs do not (e.g., a Rename-Class RFM does not enumerate all methods of which it changes the scoped name or the return-type name). Delta modules do not need to maintain functionality and thus name capture is not an issue for their safe composition; RFMs must maintain functionality and so name capture was an issue (cf. Sec. 5.3.3). Researchers evaluate products of SPLs, which involve delta modules, one by one with a constraint-based type system without composing the products [SBD11]; that is, researchers extract data about provided and required pieces of code from each delta module and test compositions of these data one by one to test whether a respective product composes safely. Researchers define a partial order for delta modules in an SPL, whereas programmers define a total order for RFMs; the total order for RFMs however is only required for the safe-composition approach, which does not exist for delta modules.

Flexible features are nonmonotonic program transformations that are used for SPLs [EVV09]. Flexible features generate, remove, and modify code in their input programs. Flexible features never cause dangling references in the programs they generate because flexible features prune the code, which they aim to add, suitably at program-generation time. As a result, the program which flexible features generate will always compile but might contain no code or might contain useless code [EVV09]. Flexible features enumerate all the pieces of code they transform, whereas RFMs do not. Safe composition of Jak-like feature modules and RFMs does only consider the complete execution of a program transformation.

Programs as graphs. Researchers represent programs as graphs and enrich these graphs to ease the implementation of refactoring types and according precondition checks [EH07, JH06]; others enrich programming languages to ease the implementation of refactoring types [SVEd09]. These approaches are foundations for precondition checks but they do not verify sequences of selectable refactorings.

Correctness of programs. Verifying properties of a program (or multiple programs) does not guarantee that this program works as intended [Smi85]. The reason is that always different aspects of the real world are neglected in the abstractions of programs. Our approach to safe composition, thus, does not verify that SPL products work as intended but only that they can be generated from feature modules without error and that they can be compiled.

7.5. Related Work on Correcting Functionality Errors of SPL Products

In Section 6.1, we discussed how programmers can detect and correct errors in functionality in SPLs which involve nonmonotonic, nonenumerative feature modules. We compared whether programmers should start at the level of feature modules or at the level of single SPL products to detect and correct errors. We concluded that starting at the level of single SPL products can be beneficial.

Connecting code at different levels of abstraction. Researchers can relate code of program transformations to code of programs which these transformations generate; these relations then allow programmers to *detect* errors of a program at the level of the transformations [ARLS05, PMS06, Sun03, vDD94, vDKT93]. In contrast to the work of these researchers, we compared for SPLs (a) error detection at the level of feature modules with (b) error detection at the level of a single SPL product. In addition, we compared for SPLs (a) error correction at the level of feature modules with (b) error correction at the level of a single SPL product. The correction of SPLs at the level of a single product has so far only been analyzed for SPLs implemented with Jak-like feature modules [Bat06] or with frameworks [CFL03], whereas we focus on SPLs implemented with nonmonotonic, nonenumerative feature modules. We analyzed how programmers can correct SPLs at the level of a single SPL product when these SPLs involve Jak-like feature modules and nonmonotonic, nonenumerative feature modules such as RFMs.

Researchers discovered problems for detecting errors in a program that was generated with aspect-oriented programming[4]; these problems are similar to some problems we discussed (they do not discuss error correction) [EAH$^+$07]: For example, when these researchers detected errors at the level of the base program (without looking at aspect code), aspects caused unexpected control-flow jumps or caused the display of incorrect code. As a result, these researchers suggest to detect errors at the level of the generated SPL product, which includes the aspect code. We discussed error detection *and* error correction for SPLs, which involve refinements (often times similar to aspects [AKKL07, KRAL07]) *and* refactorings.

Researchers keep different artifacts such as models synchronized (e.g., with user-defined bidirectional transformations) [CFH$^+$09, FGM$^+$05, HMT04, Kör10, Pie09]. We defined an algorithm to keep an SPL product synchronized with the feature

[4]Aspect-oriented programming separates code of a program into a base program and a set of aspects; aspects mainly add code to the base program [KHH$^+$01].

modules, which generate the product. The particular use case of bidirectional transformations being SPL feature modules has not been discussed before. Prior work synchronizes a model with a second model, whereas we synchronize the code of a product of an SPL (might correspond to a model) with feature modules of this SPL (corresponds to model transformations). **With respect to our discussion on refactoring multi-language programs (cf. Sec. 6.3):** Synchronization tasks in the cited approaches have not been used to trigger refactorings among artifacts of a multi-language program.

Compiler-optimized programs. Researchers detected errors in binaries of stand-alone programs at the level of the source code (i.e., code that was used to generate the binaries) [AT96, CIBR00, Fai98, FNP97, HCU92, Hen82, Str91, VRFS08, WGM08, Zel83]. In contrast, we analyzed how programmers can detect errors in feature modules at the level of a single generated program; a program which was generated from the feature modules before. In addition, we analyzed how programmers can *correct* products at the level of these single SPL products.

Refactoring during error correction. Researchers detect and invert refactorings which they executed during error correction of a stand-alone program; they do so to simplify the manual merge of that program with a legacy version of it [Dig07, DMJN07, DMJN08]; as a result, only the edits remain conflicts for the merge. We do not invert refactorings before we identify a correction but afterwards (to calculate good propagation targets). We do not propagate corrections to a stand-alone program but to feature modules of an SPL. **With respect to our discussion on reducing the generation time of SPL products (cf. Sec. 6.2):** We do not reorder, fuse, and adapt refactorings to merge versions of a program but to reduce the time a tool needs to execute these refactorings.

Maintenance through transformation. Programmers can define program transformations (maintenance deltas) to correct a program [Bax90]. Design-maintenance systems can execute these transformations consecutively to generate the corrected program [Bax90, Bax92, BP97, BPM04]. The main focus of design-maintenance systems is to record rationales for decisions which programmers consecutively make to implement a program; these records then shall help programmers to understand the finally generated program [Bax90]. In line with design-maintenance systems, we use a transformation-history concept. Design-maintenance systems *could* benefit from propagating corrections toward (legacy abstractions of) stand-alone programs [Bax90]; in contrast, we propagate corrections toward the feature modules of an SPL. **With respect to our discussion on the RFM concept (cf. Chap. 4):** In contrast to design-maintenance systems, we focus on SPLs, and SPLs do not involve maintenance deltas and rationales but selectable features. **With respect to our discussion on optimizing NFPs of SPL products (cf. Sec. 4.3.2):** Design-maintenance systems can *optimize* a program for performance goals [Bax90, Bax92];

we followed this research and gave precise guidelines for how to *configure* a program with refactorings (in RFMs) with respect to NFPs. **With respect to our discussion on reducing the generation time of SPL products (cf. Sec. 6.2):** Design-maintenance systems reorder, update, and replace transformations to integrate a new transformation with an existing sequence of program transformations (the transformation history) [Bax90]; we reorder, update, and replace sequenced transformations to reduce the time a tool needs to execute these transformations. (The fusion of transformations was indicated for design-maintenance systems but no precise rules were given.)

7.6. Related Work on Reducing the Program-Generation Time

In Section 6.2, we analyzed how programmers can reduce the time a tool needs to generate an SPL product with refactorings. To be precise, among other things, we analyzed sequences of refactorings that are executed in the course of generating an SPL product and fused refactorings where possible.

Fast execution of refactoring sequences. Researchers describe and fuse *verification* phases of sequenced refactorings to reduce the time a tool needs to execute these refactorings [CN00, KK04, Kni06, Rob99]. In contrast, we fused sequenced *transformation* phases of refactorings to reduce the time a tool needs to execute these refactorings. **With respect to our discussion on safe composition of nonmonotonic, nonenumerative feature modules (cf. Chap. 5):** We do not verify the preconditions of refactorings inside a fixed sequence of refactorings but inside a flexible sequence of refactorings (in which refactorings might be selected or might be not selected). Once we checked safe composition for an SPL, we could omit precondition checks for the involved refactorings because we then checked that no refactoring fails in any legal SPL product.

Researchers execute sequenced refactorings in parallel to reduce the time a tool needs to execute this sequence [MTR07, Rob99]. In those approaches, dependency graphs are calculated to reveal refactorings that can be executed in parallel. To avoid undesired interactions between refactorings, which execute in parallel, the researchers locked parts of the program for individual refactorings. In contrast, we proposed to analyze the program to refactor *before* we execute RFMs in parallel and proposed to detect sets of pieces of code of a program that each could contain all code a refactoring transforms; we further proposed to determine whether two refactorings interfere or whether they target different, distinct detected code sets; if they target different distinct sets, both refactorings can be executed in parallel.

Researchers relate the execution of program transformations to arrows of category theory [Bat07a, Bat07b, Bat08]. These researchers motivate that different sequences of program transformations may generate equal programs but may differ in the time a tool needs to execute either sequence. The work of these researchers is the foundation of our work because we defined a set of precise rules for how to generate sequences of refactorings, which all generate equal programs but which all may differ with respect to the time a tool needs to execute them.

Fast execution of a single program transformation. Researchers analyze how to reduce the time a tool needs to execute a tree traversal [Fai98]. Researchers optimize rewriting algorithms of composite transformations [JV01]. We also aim at executing transformations faster. However, we analyze no individual transformations and no composite transformations but sequences of transformations (refactorings).

Dependencies between program transformations. Researchers analyze dependencies between program transformations or between refactorings to calculate possible orders for their execution; orders in which either no transformation fails or only a few [MKR06, MTR07, WS97]. We detect dependencies such that we can reorder refactorings in a given sequence without changing the program this sequence generates. **With respect to our discussion on safe composition of nonmonotonic, nonenumerative feature modules (cf. Chap. 5):** We do not look for the best order to execute refactorings but we verify that refactorings execute safely for a given order (in different combinations).

Query optimization in database management systems. Algebraic and cost-based optimization of transformation sequences in SPLs is similar to algebraic and cost-based optimization of data transformations, which answer user queries in relational databases. That is, database users define queries; the database management system translates these queries into sequences of data transformations; the database management system rewrites these sequences to reduce the time a tool will need to execute them; finally, the database management system executes the rewritten sequences [Cha98, Hal76, SH99]. Database management systems rewrite sequences of data transformations without analyzing the data in the database (researchers call this *algebraic optimization*) and with analyzing the data in the database (researchers call this *cost-based optimization*) [Hal76, JK84, SC75]. We gave rules for how to rewrite user-defined sequences of *program* transformations; we proposed to rewrite these sequences without analyzing the code to transform (we called this *algebraic optimization*; cf. Sec. 6.2.1) and with analyzing the code to transform (we called this *cost-based optimization*; cf. Sec. 6.2.1).

The relationship between transformation systems and database management systems with respect to optimization of user queries is well-known [Bat04]. In line with our work, researchers transform and optimize user-defined sequences of program transformations [Bat04]. However, while they aim to change (improve) the generated program, we aim not to change the program but to reduce the time a tool needs to generate this program (to execute the sequenced transformations).

Distributed database management systems can execute transformations of a single sequence of data transformations in parallel on different computers to reduce the time they need to execute these transformations [AHY83, Cha98, JK84]. In contrast, we discussed how tools can parallelize a sequence of *program* transformations to reduce the time these tools need to execute these transformations.

7.7. Related Work on Multi-Language Support for RFMs

In Section 6.3, we reported on a study in which we applied refactorings and RFMs to multi-language programs. We observed a number of problems such as the correct prediction of method results.

Concurrent program descriptions. Researchers study the synchronous refactoring of concurrent descriptions of one program (or a piece of code therein; e.g., different UML models of one program) [BCPV07, COV06, GSMD03, Läm04, LP07, SPTJ01]. We analyze artifacts which do not describe the same piece of code of a program but which describe different pieces of code of a program. Researchers automate or analyze the synchronized transformation of related artifacts beyond refactoring [AC08, CHH05, Läm04]. Researchers automate *rename* refactorings for multi-language programs implemented with Java artifacts, SQL artifacts, and artifacts of Hibernate-like modules [TTS$^+$08]. We analyzed different languages and transformations (refactorings) than most of the above analyses did, and we found different general problems for MLR than the above analyses did.

Researchers describe an *impedance mismatch* to describe the conceptual differences between objects of an object-oriented program and tuples in a relational database when both represent the same entities [IBNW09]. Impedance mismatch is the main reason for most of our problems in implementing MLRs. Researchers identified in part the same challenges before as we identified in our study; in contrast to their work, we analyzed more than OOP languages and query languages of relational databases. While they concentrated on how to *establish* relations between artifacts, we concentrated on and faced problems for how to *maintain* these relations in refactorings.

Model integration. Researchers integrated meta-models of different languages [CJ08, GBP04, KWDE98, Mei06, SKL06, TDDN00]. The integrated meta-models include the *join* of (relevant) single-language concepts or the *intersection* of single-language concepts and related concepts, or researchers slightly *altered* at least one language. Some of these researchers refactored multi-language programs using their meta-model and some used their models only to present and understand code [KWDE98, Lin95, LTP04]. We did not use these approaches, however, because we see the following disadvantages:

(a) If we join language concepts, then every piece of code of a multi-language program implements an element in the meta-model and we can change every piece on behalf of a refactoring. The approach however does not scale well in the number of languages because the more meta-models of languages we join, the more concepts must be concerned in a single refactoring-type implementation, which is based on the meta-model.

(b) If we intersect language concepts then pieces of code, which do not implement a meta-model concept, cannot be transformed with this meta-model on behalf of a refactoring (can cause inconsistencies) [TDDN00]. The approach further does not scale well in the number of languages because the more meta-models of languages we intersect, the fewer concepts remain in the meta-model; striving to nearly useless meta-models.

(c) If we alter at least one language, then we must rewrite at least one compiler, too. A language alteration may further debase concepts that have been proven useful.

In our study we found *general* problems of MLRs (defined in RFMs); problems that were not reported so far or that were not related to refactorings so far.

Generalization of single-language refactorings. Generic refactorings [Läm02] define refactoring types in a language-independent way such that they can be instantiated for different languages. We analyzed how to refactor a single program which at the same time includes artifacts written in different languages.

Researchers explore preconditions of MLRs for programs written in languages of OOP in general, or have implemented MLRs for such programs [DLT00, KKKS08a, NSCP06, Mar05, TDDN00]. We evaluated a multi-language program which involves artifacts of OOP languages and non-OOP languages. Furthermore, researchers explore MLRs for multi-language programs that include artifacts written in OOP languages and particular other languages, such as the hypertext markup language [KKKS08a, KKKS08b, SKL06]; in these studies, all languages were based on a common intermediate language. Researchers update artifacts when database schemata changed [CHH05]. Researchers transform code without a standard database refactoring (according to [Amb03]) such that this code uses a different database management system after the transformation [CHH05, Cle09]; these researchers observed that analyses of method functionality was needed but was up to impossible [Cle09]. Researchers evolve multi-language programs beyond refactoring [VV08]. We executed *standard* refactorings (according to [Amb03, Fow99]) on a multi-language program but still faced the problem of analyzing the functionality of methods. Researchers execute refactorings on databases and update programs, which used these databases [Cle09, CMZ08], but in contrast to our work, they did not use object-relational mappers such as Hibernate and did not identify the problems we identified for MLR.

8. Conclusion

Modules are beneficial when their code can be reused. To reuse the code of a module as is, the functionality of this module is as important as its structure. If a programmer requires a module to provide some functionality but the module does not provide it, the programmer cannot reuse the module as is. If a programmer requires a module to provide some structure but the module does not provide it, the programmer cannot reuse the module as is, too. Researchers identified the dilemma that modules frequently cannot be reused because they either provide an unsuitable functionality, an unsuitable structure, or both (cf. Sec. 3.2). To tackle the problem of unsuitable functionality, researchers proposed *software product lines (SPLs)* before (i.e., sets of programs which differ in features (distinguishable program characteristics) and share features; cf. Sec. 2.2).

In this thesis, we introduced a new approach that extends techniques of SPL engineering to allow users to configure the structure (e.g., the names and allocations of pieces of code) of a module in addition to its functionality. That is, our approach to configure the structure of a module integrates well with approaches, which already allow users to configure the functionality of a module. As a result, users can now configure the functionality *and* the structure of a module. For this configuration, they do not need any knowledge regarding the implementation of a module (i.e., we provide techniques that allow users to configure the structure of a module based on the concept of features; techniques that allow users to configure functionality based on features already exist). Our approach ensures that decisions regarding the functionality of a module do not affect decisions regarding the structure of this module, and vice versa. We called this an *integrated, feature-driven configuration* of a module's functionality and structure. Henceforth, if a module provides unsuitable functionality, users can (as before) configure the module to provide suitable functionality; if a module provides unsuitable structure, users can now configure the module to provide suitable structure. Specifically, we extended techniques of SPL engineering that allow users to synthesize a module from SPL modules, which are dedicated to features (so-called *feature modules*); specifically, we extended these SPL-engineering techniques to configure the structure of the synthesized modules. After this, we generalized different techniques, which work for common modules of SPLs to work with the new SPL modules that we proposed.

8.1. Summary of the Thesis

In Chapter 3, we reported on preliminary studies in which we could not reuse a module, which had suitable functionality, because this module had an unsuitable structure. We reviewed the underlying dilemma of module scalability and summarized descriptions of this dilemma in literature. Finally, we summarized existing approaches, which tackle the dilemma of module scalability; we discovered strengths and weaknesses for all of them.

In Chapter 4, we introduced a new approach that extends techniques of SPL engineering to support an integrated, feature-driven configuration of modules. Specifically, we integrated refactorings, which are program transformations that alter the structure but not the functionality of modules, with feature modules of SPLs – we called the resulting new feature modules *refactoring feature modules (RFMs)*. We discussed implementation approaches for RFMs by analyzing refactorings with respect to other feature modules, algebraically. Finally, we discussed studies in which we used RFMs (a) to integrate modules with programs and (b) to configure nonfunctional properties of programs.

In Chapter 5, we discussed approaches of *safe composition*, which verify whether all products of an SPL can be generated without error and can be compiled. These approaches did not work when SPLs involved RFMs because the concepts of these approaches assume feature modules to monotonically remove code or to monotonically add code, and to enumerate all pieces of code the modules transform (we called these feature modules *monotonic, enumerative feature modules*). We generalized existing concepts, which were used to verify safe composition for monotonic, enumerative feature modules before [TBKC07], to cover RFMs; that is, we extended the concepts to cover feature modules that nonmonotonically add and remove code, and that do not always enumerate all transformed pieces of code (we thus called RFMs *nonmonotonic, nonenumerative feature modules*).

In Chapter 6, we discussed RFMs in three respects: First, we discussed how programmers can correct SPL products when these products were generated with RFMs. Second, we discussed how tools can reduce the time which (other) tools will need to execute sequenced RFMs (i.e., the time which these tools will need to generate an SPL product with RFMs). Finally, we discussed the refactoring of programs, which involve artifacts that are written in multiple languages.

In Chapter 7, we related all approaches presented in this thesis and all discussions of this thesis to approaches and discussions presented elsewhere.

8.2. Contribution

Most importantly, we extended an SPL approach to configure the structure of modules, we generalized an approach to verify safe composition for SPLs, and we evaluated practical issues of configuring the structure of modules with our new technique.

Extension of SPL approach to configure the structure of modules. We discussed a new approach that extends techniques of SPL engineering, which already allow SPL users to configure the functionality of a module, to support an integrated, feature-driven configuration of a module's functionality and structure. In our approach, essentially, we integrate feature modules of SPLs with refactorings (i.e., with program transformations which alter the structure but not the functionality of modules). We called this approach *refactoring feature modules (RFMs)*. We implemented a prototype to integrate monotonic, enumerative feature modules and RFMs. In studies we observed that RFMs can help to remove incompatibilities between a module and a program, and to configure nonfunctional properties of programs/modules.

Generalized approach to verify safe composition for SPLs. We generalized the concepts of an existing approach, which verifies *safe composition* of an SPL, to support SPLs with more expressive feature modules (i.e., we generalized the concepts of an approach which verifies that all products of an SPL can be generated without error and can be compiled). The concepts of existing approaches yet support only SPLs implemented with monotonic, enumerative feature modules. We generalized the concepts of one of these approaches, to support SPLs with nonmonotonic, nonenumerative feature modules.

We observed that we were hardly able to verify safe composition manually for SPLs with nonmonotonic, nonenumerative feature modules. The reason was that dependencies between nonmonotonic, nonenumerative feature modules became complex more rapidly than we expected. For example, in an SPL with nonmonotonic, nonenumerative feature modules, a single feature module may depend at the same time on that a *sequence* of feature modules executes before, that a *sequence* of feature modules does not execute before, that individual feature modules execute before, *and* that individual feature modules do not execute before. In the end, we realized that we depended ourselves on the prototype, which we developed, to verify safe composition of the studied SPLs. Nevertheless, we argue that RFMs are useful (e.g., to integrate modules with programs; cf. Sec. 4.3) and that complexity can be brought under control using tools.

Evaluation of practical issues of using RFMs. We were able to describe abstract requirements for tools that allow programmers to execute arbitrary object-oriented refactorings in monotonic, enumerative feature modules of SPLs; this analysis is important because it provides programmers of (future) refactoring tools with abstract descriptions of cases they must attend (it further helped us with our own implementation decisions). We compared approaches to detect and correct errors in the functionality of products of SPLs, which involve nonmonotonic, nonenumerative feature modules; surprisingly, the detection and correction of errors at the level of the feature modules did not outperform the detection and correction of errors at the level of a single product. We were able to prototypically implement a tool, which propagates changes from an SPL product to the feature modules of its SPL. We investigated an approach that optimizes sequenced refactorings to reduce the time a tool needs to execute these refactorings. We were able to prototypically automate the approach and to reduce the time a tool needed to execute sequences of refactorings. Finally, we investigated the refactoring of programs written with artifacts in more than one language; surprisingly, the functionality became difficult to maintain and the refactoring actions depended on tools, which the refactored program used. We were able to identify new problems of the refactoring of programs written with artifacts in more than one language, and to present a partial, automated solution for refactoring these programs.

8.3. Future Work

We did not focus on the graphical representation of RFMs so far (e.g., in user interfaces). We believe that a number of the unexpected errors we detected in our discussion on safe composition (cf. Chap. 5) were caused by the insufficient graphical support to represent RFMs. Future work could investigate such graphical concepts for RFMs. One option could be to represent the decision trees graphically, which we proposed during our investigation on safe composition. Predictable challenges to tackle then are how to represent the possibly high number of decision trees at the same time, or how to select important trees.

We did not evaluate in which cases, module reuse becomes less important than complexity, and to what extend, tools as we prototypically implemented to check for safe composition are suitable to bring complexity under control. An answer to this question certainly must take the preferences of programmers into account. A possible direction for future work would be to investigate in user studies and metrics which relate the worthiness of reusing a module (e.g., using the numbers of reused lines of code) to the complexity of reusing it (e.g., using the number of decisions exposed by a module regarding its structure). This balance between the importance of reuse and complexity certainly also could be evaluated with industrial partners.

We concentrated on standard refactorings according to catalogues and the composition of these refactorings. It remains to be shown whether nonstandard refactorings are beneficial or even required in RFMs. An answer to this research question also could answer whether nonstandard refactorings are practically important and whether to automate them pays off. To answer this question, again costs could be taken into account for (a) implementing nonstandard refactorings and for (b) saving a programmer's effort of applying these refactorings. Meaningful cost analyses again can better be undertaken with industrial partners and user studies.

We reduced the time a tool needs to execute sequenced refactorings (e.g., we fused sequenced refactorings; cf. Sec. 6.2). This goal could also be investigated for other tools that execute sequenced program transformations. That is, future work could investigate the optimization of sequences of arbitrary program transformations to reduce the time a tool needs to execute these sequences.

We refactored programs written with more than one language (cf. Sec. 6.3). One problem we faced was that functionality preservation had a different meaning for different languages. For example, the functionality of a database is only preserved when refactorings update runtime entities of this database (tuples) [Amb03], whereas functionality of a program is preserved without updating runtime entities of this program (e.g., objects for programs of object-oriented programming) [Fow99]. Future work thus could integrate different models of functionality (preservation) though we do not expect a refactoring approach which covers arbitrary languages and arbitrary refactorings.

We showed that RFMs do not fit into existing algebras of feature-oriented software development (cf. Sec. 4.2.2). Future work thus could extend the algebras to cover RFMs and other nonmonotonic or nonenumerative feature modules. Predictable challenges to tackle then for RFMs are to encode relations between pieces of code algebraically. In the same discussion, we formulated challenges that programmers face when they implement refactoring tools for SPLs; to overcome those challenges (i.e., to implement a general refactoring tool for SPLs), might also be worth future investigations.

We faced problems when we refactored artifacts of languages such as Clojure that allow variables to have no types (cf. Sec. 6.3). Specifically, we could not automatically relate pieces of code unambiguously to each other and thus could not automate a decision on whether one piece of code should be updated on behalf of the refactoring of another piece of code. Future work thus could investigate semantic models for artifacts of these languages in order to support refactoring safely.

We observed that different modules in one program might demand a single module to have different, conflicting interfaces (cf. Sec. 4.3.1). Future work might generate wrappers from RFMs to emulate alternative structures for a single module at the same time. This line of research also might investigate, what is a good balance between refactorings and wrappers. User studies again would be helpful.

A. Appendix

Further Cases of the Proof Regarding Distributivity [from KKAS11]

Case #3 $((Q_1 \cap Q_3 = Q_3), (Q_1 \cap Q_4 = \varnothing), (Q_2 \cap Q_4 \neq \varnothing))$**:**

$$\begin{array}{ll}
R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark \rangle) & \\
= ((\langle (Q_1 \cup Q_2); \checkmark \rangle \ominus Q_3) \oplus Q_4) & (4.5) \\
= (\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark \rangle \oplus Q_4) & (4.4) \\
= \langle ((Q_1 \cup Q_2) \backslash Q_3); \epsilon \rangle & (4.2)
\end{array}$$

$\lightning$

Case #4 $((Q_2 \cap Q_3 = Q_3), (Q_1 \cap Q_4 \neq \varnothing), (Q_2 \cap Q_4 = \varnothing))$**:**

$$\begin{array}{ll}
R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark \rangle) & \\
= ((\langle (Q_1 \cup Q_2); \checkmark \rangle \ominus Q_3) \oplus Q_4) & (4.5) \\
= (\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark \rangle \oplus Q_4) & (4.4) \\
= \langle ((Q_1 \cup Q_2) \backslash Q_3); \epsilon \rangle & (4.2)
\end{array}$$

$\lightning$

Case #5 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 = Q_3), Q_3 = Q_4)$**:**

$$\begin{array}{ll}
R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark \rangle) & \\
= ((\langle (Q_1 \cup Q_2); \checkmark \rangle \ominus Q_3) \oplus Q_4) & (4.5) \\
= (\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark \rangle \oplus Q_4) & (4.4) \\
= \langle (((Q_1 \cup Q_2) \backslash Q_3) \cup Q_4); \checkmark \rangle & (4.2) \\
= \langle (((Q_1 \backslash Q_3) \cup (Q_2 \backslash Q_3)) \cup Q_4); \checkmark \rangle & \\
= \langle (((Q_1 \backslash Q_3) \cup Q_4) \cup ((Q_2 \backslash Q_3) \cup Q_4)); \checkmark \rangle & \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark \rangle \bullet \langle ((Q_2 \backslash Q_3) \cup Q_4); \checkmark \rangle) & (4.3) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark \rangle \oplus Q_4) \bullet \langle ((Q_2 \backslash Q_3) \cup Q_4); \checkmark \rangle) & (4.2) \\
= (((\langle Q_1; \checkmark \rangle \ominus Q_3) \oplus Q_4) \bullet \langle ((Q_2 \backslash Q_3) \cup Q_4); \checkmark \rangle) & (4.4) \\
= (((\langle Q_1; \checkmark \rangle \ominus Q_3) \oplus Q_4) \bullet (\langle (Q_2 \backslash Q_3); \checkmark \rangle \oplus Q_4)) & (4.2) \\
= (((\langle Q_1; \checkmark \rangle \ominus Q_3) \oplus Q_4) \bullet ((\langle Q_2; \checkmark \rangle \ominus Q_3) \oplus Q_4)) & (4.4) \\
= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark \rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark \rangle)) & (4.5)
\end{array}$$

$\square$

Case #6 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 = Q_3), Q_3 \neq Q_4, ((Q_1 \backslash Q_3) \cap Q_4 \neq \varnothing))$**:**

$$\begin{array}{ll}
(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & \\
= (((\langle Q_1; \checkmark\rangle \ominus Q_3) \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.5) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark\rangle \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.4) \\
= (\langle (Q_1 \backslash Q_3); \epsilon\rangle \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.2) \\
= (\langle (Q_1 \backslash Q_3); \epsilon\rangle \bullet \langle Q_5; e_1\rangle) & (R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle) = \langle Q_5; e_1\rangle) \\
= \langle (Q_1 \backslash Q_3); \epsilon \rangle & (4.3)
\end{array}$$

$\lightning$

Case #7 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 = Q_3), Q_3 \neq Q_4, ((Q_1 \backslash Q_3) \cap Q_4 = \varnothing), ((Q_2 \backslash Q_3) \cap Q_4 \neq \varnothing))$**:**

$$\begin{array}{ll}
(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & \\
= (((\langle Q_1; \checkmark\rangle \ominus Q_3) \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.5) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark\rangle \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.2) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet ((\langle Q_2; \checkmark\rangle \ominus Q_3) \oplus Q_4)) & (4.5) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet (\langle (Q_2 \backslash Q_3); \checkmark\rangle \oplus Q_4)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet \langle (Q_2 \backslash Q_3); \epsilon\rangle) & (4.2) \\
= \langle ((Q_1 \backslash Q_3) \cup Q_4); \epsilon \rangle & (4.3)
\end{array}$$

$\lightning$

Case #8 $((Q_1 \cap Q_3 = Q_3), (Q_2 \cap Q_3 = Q_3), Q_3 \neq Q_4, ((Q_1 \backslash Q_3) \cap Q_4 = \varnothing), ((Q_2 \backslash Q_3) \cap Q_4 = \varnothing))$**:**

$$\begin{array}{ll}
(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & \\
= (((\langle Q_1; \checkmark\rangle \ominus Q_3) \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.5) \\
= ((\langle (Q_1 \backslash Q_3); \checkmark\rangle \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle)) & (4.2) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet ((\langle Q_2; \checkmark\rangle \ominus Q_3) \oplus Q_4)) & (4.5) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet (\langle (Q_2 \backslash Q_3); \checkmark\rangle \oplus Q_4)) & (4.4) \\
= (\langle ((Q_1 \backslash Q_3) \cup Q_4); \checkmark\rangle \bullet \langle ((Q_2 \backslash Q_3) \cup Q_4); \checkmark\rangle) & (4.2) \\
= \langle (((Q_1 \backslash Q_3) \cup Q_4) \cup ((Q_2 \backslash Q_3) \cup Q_4)); \checkmark\rangle & (4.3) \\
= \langle (((Q_1 \backslash Q_3) \cup (Q_2 \backslash Q_3)) \cup Q_4); \checkmark\rangle & \\
= \langle (((Q_1 \cup Q_2) \backslash Q_3) \cup Q_4); \checkmark\rangle & \\
= (\langle ((Q_1 \cup Q_2) \backslash Q_3); \checkmark\rangle \oplus Q_4) & (4.2) \\
= ((\langle (Q_1 \cup Q_2); \checkmark\rangle \ominus Q_3) \oplus Q_4) & (4.4) \\
= R_{Q_3 \mapsto Q_4}(\langle (Q_1 \cup Q_2); \checkmark\rangle) & (4.5)
\end{array}$$

$\square$

Case #9 ($(Q_2 \cap Q_3 \neq Q_3)$):

$(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle))$

$= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet ((\langle Q_2; \checkmark\rangle \ominus Q_3) \oplus Q_4))$ (4.5)

$= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet (\langle Q_2; \epsilon\rangle \oplus Q_4))$ (4.4)

$= (R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet \langle Q_2; \epsilon\rangle)$ (4.2)

$= (\langle Q_5; e_1\rangle \bullet \langle Q_2; \epsilon\rangle)$ $(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) = \langle Q_5; e_1\rangle)$

$= \langle Q_5; \epsilon\rangle$ (4.3)

↯

Case #10 ($(Q_1 \cap Q_3 \neq Q_3)$):

$(R_{Q_3 \mapsto Q_4}(\langle Q_1; \checkmark\rangle) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle))$

$= (((\langle Q_1; \checkmark\rangle \ominus Q_3) \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle))$ (4.5)

$= ((\langle Q_1; \epsilon\rangle \oplus Q_4) \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle))$ (4.4)

$= (\langle Q_1; \epsilon\rangle \bullet R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle))$ (4.2)

$= (\langle Q_1; \epsilon\rangle \bullet \langle Q_5; e_1\rangle)$ $(R_{Q_3 \mapsto Q_4}(\langle Q_2; \checkmark\rangle) = \langle Q_5; e_1\rangle)$

$= \langle Q_1; \epsilon\rangle$ (4.3)

↯

Bibliography

[AC08] M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering II (GTTSE)*, pages 3–46. Springer Verlag, 2008.

[ADT07] F.I. Anfurrutia, O. Díaz, and S. Trujillo. On refining XML artifacts. In *Proceedings of the International Conference on Web Engineering (ICWE)*, pages 473–478. Springer Verlag, 2007.

[AGM$^+$06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 201–210. ACM Press, 2006.

[AHY83] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering (TSE)*, 9(1):57–68, 1983.

[AKB08] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170. ACM Press, 2008.

[AKGL10] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering – An International Journal*, 17(3):251–300, 2010.

[AKKL07] S. Apel, C. Kästner, M. Kuhlemann, and T. Leich. Pointcuts, advice, refinements, and collaborations: Similarities, differences, and synergies. *Innovations in Systems and Software Engineering (ISSE)*, 3(4):281–289, 2007.

[AKL06] S. Apel, M. Kuhlemann, and T. Leich. Generic feature modules: Two-staged program customization. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, pages 127–132. INSTICC Press, 2006.

[AKL09] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the*

International Conference on Software Engineering (ICSE), pages 221–231. IEEE Computer Society, 2009.

[AKLS07] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect refinement - Unifying AOP and stepwise refinement. *Journal of Object Technology (JOT)*, 6(9):13–33, 2007. Special Issue. TOOLS EUROPE 2007.

[ALMK10] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[ALS08] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[Amb03] S.W. Ambler. *Agile database techniques: Effective strategies for the agile software developer.* John Wiley & Sons, Inc., 2003.

[APH$^+$08] N. Ayewah, W. Pugh, D. Hovemeyer, J.D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[ARLS05] S. Apel, M. Rosenmüller, T. Leich, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140. Springer Verlag, 2005.

[Aßm98] U. Aßmann. Optimix - A tool for rewriting and optimizing programs. In *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools*, pages 307–318. World Scientific Publishing, 1998.

[AT96] A.-R. Adl-Tabatabai. *Source-level debugging of globally optimized code.* PhD thesis, Carnegie Mellon University of Pittsburgh, USA, 1996.

[BAS08] D. Batory, M. Azanza, and J. Saraiva. The objects and arrows of computational design. In *Proceedings of the Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 1–20. Springer Verlag, 2008.

[Bat04] D. Batory. The road to Utopia: A future for generative programming. In *Proceedings of the Seminar on Domain-Specific Program Generation*, pages 211–250. Springer Verlag, 2004.

[Bat05] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 7–20. Springer Verlag, 2005.

[Bat06] D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *Proceedings of the Summer School on Generative and*

Transformational Techniques in Software Engineering (GTTSE), pages 3–35. Springer Verlag, 2006.

[Bat07a] D. Batory. From implementation to theory in product synthesis. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, volume 42, pages 135–136. ACM Press, 2007.

[Bat07b] D. Batory. A modeling language for program design and synthesis. In *Proceedings of the Lipari Summer School on Advances in Software Engineering*, pages 39–58. Springer Verlag, 2007.

[Bat07c] D. Batory. Program refactoring, program synthesis, and model-driven development. In *Proceedings of the Conference on Compiler Construction (CC)*, pages 156–171. Springer Verlag, 2007.

[Bat08] D. Batory. Using modern mathematics as an FOSD modeling language. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 35–44. ACM Press, 2008.

[Bat09] D. Batory. On the importance and challenges of FOSD, 2009. Keynote at Workshop on Feature-Oriented Software Development [Available online: `http://www.infosun.fim.uni-passau.de/cl/staff/apel/-FOSD2009/BatoryFOSDKeynote2.pdf`; accessed: July 16,2011].

[Bax90] I.D. Baxter. *Transformational maintenance by reuse of design histories*. PhD thesis, University of California at Irvine, USA, 1990.

[Bax92] I.D. Baxter. Design maintenance systems. *Communications of the ACM (CACM)*, 35(4):73–89, 1992.

[BB08] D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *Journal of Universal Computer Science (J.UCS)*, 14(12):2059–2082, 2008.

[BCK06] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice.* Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2006.

[BCL10] T.T. Bartolomei, K. Czarnecki, and R. Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE Computer Society, 2010.

[BCLvdS10] T.T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API migration for two XML APIs. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 42–61. Springer Verlag, 2010.

[BCPV07] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for XML and SQL. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 290–304. Springer Verlag, 2007.

[BCS00] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 227–247. Kluwer Academic Publishers, 2000.

[BCVM02] A. Bryant, A. Catton, K. De Volder, and G.C. Murphy. Explicit programming. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, pages 10–18. ACM Press, 2002.

[BDN05] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189. ACM Press, 2005.

[BEA03] BEA Systems, Inc. *JSR-000173 Streaming API for XML Specification 1.0*, 2003. [Available online: `http://jcp.org/aboutJava/communityprocess/-final/jsr173/`; accessed: July 16,2011].

[BG97] D. Batory and B.J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering (TSE)*, 23(2):67–82, 1997.

[Big98] T.J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[Big04] T.J. Biggerstaff. A new architecture for transformation-based generators. *IEEE Transactions on Software Engineering (TSE)*, 30(12):1036–1054, 2004.

[BKS10] D. Bruns, V. Klebanov, and I. Schaefer. Verification of software product lines: Reducing the effort with delta-oriented slicing and proof reuse. In *Proceedings of the Conference on Formal Verification of Object-oriented Software (FoVeOOS)*, pages 61–75. Springer Verlag, 2010.

[BKVV08] M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming (SCP)*, 72(1-2):52–70, 2008.

[BLO07] A. Bergel, C. Lewerentz, and L. O'Brien. Classboxes: Supporting unanticipated variation points in the source code. In *Proceedings of the Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 8–13. Lancaster University, UK, 2007.

[BMMB00] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson. Object-oriented framework-based software development: Problems and experiences. *ACM Computing Surveys (CSUR)*, 32(1 es):3, 2000.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[Bos98] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming (JOOP)*, 11(2):18–32, 1998.

[BOT07] G. Botterweck, L. O'Brien, and S. Thiel. Model-driven derivation of product architectures. In *Proceedings of the Conference on Automated Software Engineering (ASE)*, pages 469–472. ACM Press, 2007.

[BP97] I.D. Baxter and C.W. Pidgeon. Software change through design maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 250–259. IEEE Computer Society, 1997.

[BPM04] 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. IEEE Computer Society, 2004.

[BS07] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-04-66, University of Texas at Austin, USA, 2007.

[BSR04] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE)*, pages 191–199. ACM Press, 1993.

[BSTR07] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134. Lero-University of Limerick, Ireland, 2007.

[BTF05] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 265–279. ACM Press, 2005.

[BTR05] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 491–503. Springer Verlag, 2005.

[CAK06] K. Czarnecki, M. Antkiewicz, and C.H.P. Kim. Multi-level customization in application engineering. *Communications of the ACM (CACM)*, 49(12):60–65, 2006.

[CB74] D.D. Chamberlin and R.F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the Workshop on Data description, Access and Control*, pages 249–264. ACM Press, 1974.

[CBS$^+$07] F. Calheiros, P. Borba, S. Soares, V. Nepomuceno, and V. Alves. Product line variability refactoring tool. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 33–34. TU Berlin, Germany, 2007.

[CDCv03] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek. Refactoring product line architectures. In *Proceedings of the Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, pages 23–26,

2003. [Available online: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.1385&rep=rep1&type=pdf`; accessed: July 26,2011].

[CE99a] K. Czarnecki and U.W. Eisenecker. Components and generative programming (invited paper). In *Proceedings of the European Software Engineering Conference/Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 2–19. Springer Verlag, 1999.

[CE99b] K. Czarnecki and U.W. Eisenecker. Synthesizing objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 18–42. Springer Verlag, 1999.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative programming: Methods, tools, and applications.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[CFH$^+$09] K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 260–283. Springer Verlag, 2009.

[CFL03] M. Cortes, M. Fontoura, and C. Lucena. Using refactoring and unification rules to assist framework evolution. *ACM Software Engineering Notes (SEN)*, 28(6):1–1, 2003.

[CG94] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 397–408. ACM Press, 1994.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 34–43. ACM Press, 1998.

[CHH05] A. Cleve, J. Henrard, and J.-L. Hainaut. Co-transformations in information system reengineering. In *Proceedings of the Workshop on Metamodels, Schemas, and Grammars for Reverse Engineering (ATEM)*, pages 5–15. Elsevier B. V., 2005.

[Chr04] H.B. Christensen. Frameworks: Putting design patterns into perspective. *ACM SIGCSE Bulletin*, 36(3):142–145, 2004.

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM (CACM)*, 40(10):70–77, 1997.

[CIBR00] D.L. Curreri, A.K. Iyengar, R.A. Biesele, and M.A. Ruscetta. Debugging optimized code using data change points, 2000. US patent #6,091,896.

[CJ08] N. Chen and R. Johnson. Toward refactoring in a polyglot world: Extending automated refactoring support across Java and XML. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 1–4. ACM Press, 2008.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493, 1994.

[CK02] P. Clements and C. Krueger. Point/counterpoint: Being proactive pays off/eliminating the adoption barrier. *IEEE Software*, 19(4):28–31, 2002.

[CL01] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294. ACM Press, 2001.

[Cle09] A. Cleve. *Program analysis and transformation for data-intensive system evolution*. PhD thesis, University of Namur, Belgium, 2009.

[CM07] J.S. Cuadrado and J.G. Molina. A phasing mechanism for model transformation languages. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1020–1024. ACM Press, 2007.

[CMZ08] C.A. Curino, H.J. Moon, and C. Zaniolo. Graceful database schema evolution: The PRISM workbench. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):761–772, 2008.

[CN00] M. Ó Cinnéide and P. Nixon. Composite refactorings for Java programs. In *Proceedings of the Workshop on Formal Techniques for Java Programs (FTfJP)*, pages 129–135, 2000.

[CN06] P. Clements and L. Northrop. *Software product lines: Practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[COV06] A. Cunha, J. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Proceedings of the Symposium on Formal Methods (FM)*, pages 284–299. Springer Verlag, 2006.

[CP06] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

[CR96] A. Colmerauer and P. Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367. ACM Press, 1996.

[CRB04] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, University of Lancaster, UK, 2004.

[CW07] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 23–34. IEEE Computer Society, 2007.

[DC94] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. *ACM SIGPLAN Lisp Pointers*, VII(3):273–282, 1994.

[DCB09] B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM Press, 2009.

[Dev09] P. Deva. Explore refactoring functions in Eclipse JDT, 2009. [Available online: `https://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring`; accessed: July 16,2011].

[Dig07] D. Dig. *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 2007.

[Dij69] E.W. Dijkstra. Structured programming (EWD268), 1969. [Available online: `http://userweb.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF`; accessed: July 16,2011].

[Dij82] E.W. Dijkstra. On the role of scientific thought (EWD447). In *Selected writings on computing: A personal perspective*, pages 60–66. Springer Verlag, 1982.

[DK76] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121. ACM Press, 1976.

[DLT00] S. Ducasse, M. Lanza, and S. Tichelaar. MOOSE: An extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Symposium on Constructing Software Engineering Tools (COSET)*, pages 24–30. University of Limerick, Ireland, 2000.

[DMJN07] D. Dig, K. Manzoor, R. Johnson, and T.N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 427–436. IEEE Computer Society, 2007.

[DMJN08] 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 (TSE)*, 34(3):321–335, 2008.

[DNMJ08] 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. ACM Press, 2008.

[EAH$^+$07] M. Eaddy, A.V. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Proceedings of the Symposium on Software Composition (SC)*, pages 200–215. Springer Verlag, 2007.

[EBLSP10] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat. Variability in time – Product line variability and evolution revisited. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 131–137. University of Duisburg-Essen, Germany, 2010.

[Eff11] S. Efftinge. *openArchitectureWare 4.1: Check – Validation language*. openArchitectureWare.org, 2011. [Available online: `http://www.openarchitectureware.org/pub/documentation/4.1/r30_checkReference.pdf`; accessed: July 16,2011].

[EH07] T. Ekman and G. Hedin. The JastAdd system – Modular extensible compiler construction. *Science of Computer Programming (SCP)*, 69:14–26, 2007.

[EM04] D.R. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 191–210. Springer Verlag, 2004.

[ESWH04] S.H. Edwards, M. Sitaraman, B.W. Weide, and J. Hollingsworth. Contract-checking wrappers for C++ classes. *IEEE Transactions on Software Engineering (TSE)*, 30(11):794–810, 2004.

[EVV09] P. Ebraert, J. Vallejos, and Y. Vandewoude. Flexible features: Making feature modules more reusable. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1963–1970. ACM Press, 2009.

[Fai98] R.E. Faith. *Debugging programs after structure-changing transformation.* PhD thesis, University of North Carolina at Chapel Hill, USA, 1998.

[FCDR95] I.R. Forman, M.H. Conner, S.H. Danforth, and L.K. Raper. Release-to-release binary compatibility in SOM. *ACM SIGPLAN Notices*, 30(10):426–438, 1995.

[FGM$^+$05] J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 233–246. ACM Press, 2005.

[FKK07] R.M. Fuhrer, M. Keller, and A. Kieżun. Advanced refactoring in the Eclipse JDT: Past, present, and future. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 30–31. TU Berlin, Germany, 2007.

[FNP97] R.E. Faith, L.S. Nyland, and J.F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, pages 243–255. USENIX Association Berkeley, USA, 1997.

[For08] N. Ford. *The productive programmer.* O'Reilly & Associates, Inc., 2008.

[Fow99] M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[FTK04] R. Fuhrer, F. Tip, and A. Kieżun. Advanced refactorings in Eclipse. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, page 8. ACM Press, 2004.

[GA07] V. Gasiunas and I. Aracic. Dungeon: A case study of feature-oriented programming with virtual classes. In *Proceedings of the Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 32–37. Lancaster University, UK, 2007.

[GBP04] M. Grechanik, D. Batory, and D.E. Perry. Design of large-scale polylingual systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 357–366. IEEE Computer Society, 2004.

[Ger09] J. Gerdes Jr. User interface migration of Microsoft Windows applications. *Software Maintenance and Evolution: Research and Practice*, 21(3):171–187, 2009.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[Gho04] D. Ghosh. Generics in Java and C++: A comparative model. *ACM SIGPLAN Notices*, 39:40–47, 2004.

[GJ05] I. Godil and H.-A. Jacobsen. Horizontal decomposition of Prevayler. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100. IBM Press, 2005.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2005.

[Gog96] J.A. Goguen. Parameterized programming and software architecture. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 2–10. IEEE Computer Society, 1996.

[GP09] S. Götz and M. Pukall. On performance of delegation in Java. In *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pages 1–6. ACM Press, 2009.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[Gri00] M.L. Griss. Implementing product-line features by composing aspects. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 271–289. Kluwer Academic Publishers, 2000.

[Gro04] Object Management Group. *Common object request broker architecture: Core specification (Version 3.0.3)*, 2004. [Available online: `http://www.omg.org/-spec/CORBA/3.0.3/PDF`; accessed: July 14,2011].

[GSF$^+$05] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–14. ACM Press, 2005.

[GSMD03] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of the Conference on the Unified Modeling Language, Modeling Languages and Applications (UML)*, pages 144–158. Springer Verlag, 2003.

[HA09] O. Hummel and C. Atkinson. The Managed Adapter pattern: Facilitating glue code generation for component reuse. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 211–224. Springer Verlag, 2009.

[Hal76] P.A.V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1976.

[HC02] F. Hunleth and R. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 38–45. ACM Press, 2002.

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43. ACM Press, 1992.

[Hen82] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, 1982.

[Hen08] M. Henning. The rise and fall of CORBA. *Communications of the ACM (CACM)*, 51(8):52–57, 2008.

[Her08] S. Herrmann. Gradual encapsulation. *Journal of Object Technology (JOT)*, 7(9):47–68, 2008.

[HKA10] F. Heidenreich, J. Kopcsek, and U. Aßmann. Safe composition of transformations. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 108–122. Springer Verlag, 2010.

[HKR$^+$07] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. An algebraic view on the semantics of model composition. In *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, pages 99–113. Springer Verlag, 2007.

[HM07] S. Herrmann and M. Mosconi. Integrating Object Teams and OSGi: Joint efforts for superior modularity. *Journal of Object Technology (JOT)*, 6(9):105–125, 2007.

[HMT04] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 178–189. ACM Press, 2004.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 411–428. ACM Press, 1993.

[Höl93] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 36–56. Springer Verlag, 1993.

[HRRS11] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Delta modeling for software architectures. In *Proceedings of the Workshop on Model-Based Development of Embedded Systems (MBEES)*, pages 1–10. fortiss GmbH, 2011.

[HS08] S.S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 79–89. ACM Press, 2008.

[HZS05] S.S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 309–326. Springer Verlag, 2005.

[HZS07] S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424. Springer Verlag, 2007.

[IBNW09] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *Proceedings of the Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, pages 36–43. IEEE Computer Society, 2009.

[IKI04] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 178–187. IEEE Computer Society, 2004.

[Int07] Intermetrics, Inc. *Ada reference manual, ISO/IEC 8652:2007(E)*, 3rd edition, 2007. [Available online: `http://www.adaic.com/standards/05rm/-RM-Final.pdf`; accessed: July 14,2011].

[JF88] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP)*, 1(2):22–35, 1988.

[JH06] N. Juillerat and B. Hirsbrunner. FOOD: An intermediate model for automated refactoring. In *Proceedings of the Conference on Software Methodologies, Tools and Techniques (SoMeT)*, pages 452–461. IOS Press, 2006.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.

[JMS07] J. Järvi, M.A. Marcus, and J.N. Smith. Library composition and adaptation using C++ concepts. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 73–82. ACM Press, 2007.

[Jon03] S.P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[JS03] S. Jarzabek and L. Shubiao. Eliminating redundancies with a "Composition With Adaptation" meta-programming technique. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE)*, pages 237–246. ACM Press, 2003.

[JV01] P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 57:144–162, 2001.

[KA08] C. Kästner and S. Apel. Type-checking software product lines - A formal approach. In *Proceedings of the Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2008.

[KAB07] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 223–232. IEEE Computer Society, 2007.

[KAK08] 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. ACM Press, 2008.

[KAK09] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

[KARL08] M. Kuhlemann, S. Apel, M. Rosenmüller, and R.E. Lopez-Herrejon. A multi-paradigm study of crosscutting modularity in design patterns. In *Proceedings of the Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 121–140. Springer Verlag, 2008.

[KAT$^+$09] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proceedings of the Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 175–194. Springer Verlag, 2009.

[KBA08] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. Technical Report FIN-15-2008, University of Magdeburg, Germany, 2008.

[KBA09] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 106–115. Springer Verlag, 2009.

[KBK09] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 177–186. ACM Press, 2009.

[KCH$^+$90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University of Pittsburgh, USA, 1990.

[KH98] R. Keller and U. Hölzle. Binary component adaptation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 307–329. Springer Verlag, 1998.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer Verlag, 2001.

[KK04] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming (SCP)*, 52(1-3):9–51, 2004.

[KKA10] M. Kuhlemann, C. Kästner, and S. Apel. Reducing code replication in delegation-based Java programs. In *Java Software and Embedded Systems*, pages 171–183. Nova Science Publishers, Inc., 2010.

[KKAS11] M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. An algebra for refactoring and feature-oriented programming. Technical Report FIN-006-2011, University of Magdeburg, Germany, 2011.

[KKB07] C. Kästner, M. Kuhlemann, and D. Batory. Automating feature-oriented refactoring of legacy applications. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 62–63. TU Berlin, Germany, 2007.

[KKB08] C.H.P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–34. ACM Press, 2008.

[KKKS08a] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad. Cross language refactoring for Eclipse plug-ins. In *Proceedings of the Workshop on Refactoring Tools (WRT)*, pages 1–4. ACM Press, 2008.

[KKKS08b] M. Klenk, R. Kleeb, M. Kempf, and P. Sommerlad. Refactoring support for the Groovy-Eclipse plug-in. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 727–728. ACM Press, 2008.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer Verlag, 1997.

[KLS10a] M. Kuhlemann, L. Liang, and G. Saake. Algebraic and cost-based optimization of refactoring sequences. In *Proceedings of the Workshop on Model-Driven Product Line Engineering (MDPLE)*, pages 37–48. CEUR-WS.org, 2010.

[KLS10b] M. Kuhlemann, L. Liang, and G. Saake. Algebraic and cost-based optimization of refactoring sequences. Technical Report FIN-005-2010, University of Magdeburg, Germany, 2010.

[Kni99] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 351–366. Springer Verlag, 1999.

[Kni06] G. Kniesel. A logic foundation for program transformations. Technical Report IAI-TR-2006-1, University of Bonn, Germany, 2006.

[Kör10] A.-T. Körtgen. New strategies to resolve inconsistencies between models of decoupled tools. In *Proceedings of the Workshop of Living with Inconsistencies in Software Development (LWI)*, pages 21–38. CEUR-WS.org, 2010.

[KPRS01] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proceedings of the Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 57–69. Springer Verlag, 2001.

[KPS08] J. Kim, S. Park, and V. Sugumaran. DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines. *Journal of Systems and Software (JSS)*, 81(1):37–55, 2008.

[KR05] H. Krahn and B. Rumpe. Techniques enabling generator refactoring. Technical Report TR-CCTC/DI-36, Universidade do Minho of Braga, Portugal, 2005.

[KRAL07] M. Kuhlemann, M. Rosenmüller, S. Apel, and T. Leich. On the duality of aspect-oriented and feature-oriented design patterns. In *Proceedings of the Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 5. ACM Press, 2007.

[KRT97] A. Karhinen, A. Ran, and T. Tallgren. Configuring designs for reuse. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 701–710. ACM Press, 1997.

[Kru06] C.W. Krueger. New methods in software product line practice. *Communications of the ACM (CACM)*, 49(12):37–40, 2006.

[KS06] M. Keith and M. Schincariol. *Pro EJB 3: Java persistence API (Pro)*. Apress, 2006.

[KS10a] M. Kuhlemann and M. Sturm. Debugging product line programs. Technical Report FIN-006-2010, University of Magdeburg, Germany, 2010.

[KS10b] M. Kuhlemann and M. Sturm. Patching product line programs. In *Proceedings of the Workshop on Feature-Oriented Software Development (FOSD)*, pages 33–40, 2010.

[KSA10] M. Kuhlemann, N. Siegmund, and S. Apel. Using collaborations to encapsulate features? An explorative study. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 139–142. University of Duisburg-Essen, Germany, 2010.

[KWDE98] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 135–143. IEEE Computer Society, 1998.

[Lad03] R. Laddad. *AspectJ in action: Practical aspect-oriented programming.* Manning Publications Co., 2003.

[Läm02] R. Lämmel. Towards generic refactoring. In *Proceedings of the Workshop on Rule-Based Programming (RULE)*, pages 15–28. ACM Press, 2002.

[Läm04] R. Lämmel. Coupled software transformations (extended abstract). In *Proceedings of the Workshop on Software Evolution Transformations (SET)*, pages 31–35, 2004.

[LB04] J. Liu and D. Batory. Automatic remodularization and optimized synthesis of product-families. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 379–395. Springer Verlag, 2004.

[LBL06] 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. ACM Press, 2006.

[LGS09] Y.A. Liu, M. Gorbovitski, and S.D. Stoller. A language and framework for invariant-driven transformations. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 55–64. ACM Press, 2009.

[LHB01] R.E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Springer Verlag, 2001.

[LHB05] R.E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2005. [Available online: `http://www.cs.utexas.edu/ftp/-predator/SPLAT2005.pdf`; accessed: July 26,2011].

[LHBC05] R.E. Lopez-Herrejon, D. Batory, and W.R. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 169–194. Springer Verlag, 2005.

[Li06] H. Li. *Refactoring Haskell programs.* PhD thesis, University of Kent at Canterbury, UK, 2006.

[Lia99] S. Liang. *The Java^{TM}native interface: Programmer's guide and specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[Lia10] L. Liang. Optimizing sequences of refactorings. Master thesis, University of Magdeburg, Germany, March 2010. [Available online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/-auto/thesisLiang.pdf`; accessed: July 16,2011].

[Lin95] P.K. Linos. PolyCARE: A tool for re-engineering multi-language program integrations. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 338–341. IEEE Computer Society, 1995.

[LJWF02] D. Lacey, N.D. Jones, E. Van Wyk, and C.C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 283–294. ACM Press, 2002.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University of Boston, USA, 1999.

[LO06] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170. ACM Press, 2006.

[LP07] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 151–162. IEEE Computer Society, 2007.

[LSH98] J. Laski, W. Stanley, and J. Hurst. Dependency analysis of Ada programs. In *Proceedings of the Conference on Ada (SIGAda)*, pages 263–275. ACM Press, 1998.

[LTP04] T.C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl middle metamodel: A schema for reverse engineering. In *Proceedings of the Workshop on Meta-Models and Schemas for Reverse Engineering (ateM)*, pages 7–18. Elsevier B. V., 2004.

[Lyn06] I. Lynagh. An algebra of patches, 2006. [Available online: `http://-urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf`; accessed: July 16,2011].

[Mar05] R. Marticorena. Analysis and definition of a language independent refactoring catalog. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 8–16. Springer Verlag, 2005.

[MB97] M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 203–214. IEEE Computer Society, 1997.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, 2:308–320, 1976.

[MEDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[Mei06] E. Meijer. There is no impedance mismatch (language integrated query in Visual Basic 9). In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 710–711. ACM Press, 2006.

[Mey92] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mey97] B. Meyer. *Object-oriented software construction.* Prentice Hall PTR, 2nd edition, 1997.

[MF04] M.P. Monteiro and J.M.L. Fernandes. Object-to-aspect refactorings for feature extraction. In *Industry track of the Conference on Aspect-Oriented Software Development (AOSD)*, 2004. [Available online: `http://aosd.net/-2004/archive/Monteiro.pdf`, accessed: July 14,2011].

[MF05] M.P. Monteiro and J.M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, 2005.

[MHM09] P. McGachey, A.L. Hosking, and J.E.B. Moss. Classifying Java class transformations for pervasive virtualized access. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 75–84. ACM Press, 2009.

[MHQB05] 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 282–291. ACM Press, 2005.

[MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the Conference on Compiler Construction (CC)*, pages 46–60. Springer Verlag, 2003.

[MKR06] T. Mens, G. Kniesel, and O. Runge. Transformation dependency analysis - A comparison of two approaches. In *Proceedings of Langages et Modèles à Objets (LMO)*, pages 167–184. Hermes Science Publishing, 2006.

[ML98] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–116. ACM Press, 1998.

[MMZ$^+$01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Conference on Design Automation (DAC)*, pages 530–535. ACM Press, 2001.

[MO02] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 52–67. ACM Press, 2002.

[Mos11] S. Moschinski. The impact of refactoring on non-functional software properties. Master thesis, University of Magdeburg, Germany, March 2011. [Available online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisMoschinski.pdf`; accessed: July 16,2011].

[MPY$^+$04] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 459–468. IEEE Computer Society, 2004.

[MSL00] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 325–356. Kluwer Academic Publishers, 2000.

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering (TSE)*, 30:126–139, 2004.

[MTR07] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling (SoSyM)*, 6(3):269–285, 2007.

[MWC09] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 231–240. Carnegie Mellon University of Pittsburg, USA, 2009.

[Mye88] B.A. Myers. A taxonomy of window manager user interfaces. *IEEE Computer Graphics and Applications*, 8(5):65–84, 1988.

[Nat06] National University of Singapore and Netron Inc. *XML-based variant configuration language (XVCL): Specification version 2.10*, 2006. [Available online: `http://sourceforge.net/projects/fxvcl/files/XVCL%20Specification/Version%202.10/XVCL_spec_2_10.pdf`, accessed: July 14,2011].

[Nov95] 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.

[Nov97] G.S. Novak Jr. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering*, 23(7):401–417, 1997.

[NSCP06] C. López Nozal, R. Marticorena Sánchez, Y. Crespo, and F.J. Pérez. Towards a language independent refactoring framework. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, pages 165–170. INSTICC Press, 2006.

[Ode10] M. Odersky. *The Scala language specification (version 2.8, November 9,2010)*, 2010. [Available online: `http://www.scala-lang.org/`; accessed: November 11,2010].

[OG09] E.M. Olimpiew and H. Gomaa. Reusable model-based testing. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 76–85. Springer Verlag, 2009.

[OMR10] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 196–210. Springer Verlag, 2010.

[Opd92] W.F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, USA, 1992.

[Ora05] Oracle Corporation. *Oracle® C++ call interface, Programmer's guide, 10g release 2 (10.2), B14294-02*, 2005. [Available online: `http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14294.pdf`, accessed: July 14,2011].

[OT00] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *Proceedings of the Workshop on Aspects and Dimensions of Concern*, 2000. [Available online: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.4713&rep=rep1&type=pdf`; accessed: July 20,2011].

[OT01] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM (CACM)*, 44(10):43–50, 2001.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM (CACM)*, 15(12):1053–1058, 1972.

[Par76] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering (TSE)*, 2(1):1–9, 1976.

[Par78] D.L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 264–277. IEEE Computer Society, 1978.

[PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software product line engineering: Foundations, principles and techniques*. Springer Verlag, 2005.

[Pér08] J. Pérez. Enabling refactoring with HTN planning to improve the design smells correction activity. In *BElgian-NEtherlands software eVOLution workshop (BENEVOL)*, pages 48–51, 2008.

[Pie09] B.C. Pierce. Foundations for bidirectional programming. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 1–3. Springer Verlag, 2009.

[PMJ$^+$05] M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[PMS06] Z. Porkoláb, J. Mihalicza, and Á. Sipos. Debugging C++ template metaprograms. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 255–264. ACM Press, 2006.

[PR01] J. Philipps and B. Rumpe. Roots of refactoring. In *Proceedings of the Workshop on Behavioral Semantics*, pages 187–199. Northeastern University of Boston, USA, 2001.

[PR03] J. Philipps and B. Rumpe. Refactoring of programs and specifications. In *Practical Foundations of Business System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Pre97] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer Verlag, 1997.

[PRT08] J. Pérez, O. Runge, and G. Taentzer. Specifying and analyzing program refactorings with AGG. In *Proceedings of the Workshop on Graph-Based Tools: The Contest (GraBaTs)*, 2008. [Available online: `http://fots.ua.ac.be/-events/grabats2008/submissions/grabats2008_submission_23.pdf`; accessed: July 26,2011].

[RBJ97] D.B. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and practice of object systems - Special issue object-oriented software evolution and re-engineering*, 3:253–263, 1997.

[RC07] C.K. Roy and J.R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University of Kingston, Canada, 2007.

[RCD09] C. Reichenbach, D. Coughlin, and A. Diwan. Program metamorphosis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 394–418. Springer Verlag, 2009.

[Rob99] D.B. Roberts. *Practical analysis for refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, USA, 1999.

[RPB09] P.O. Rossel, D. Perovich, and M.C. Bastarrica. Reuse of architectural knowledge in SPL development. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 191–200. Springer Verlag, 2009.

[RS10] M. Rosenmüller and N. Siegmund. Automating the configuration of multi software product lines. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. University of Duisburg-Essen, Germany, 2010.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer Verlag, 1998.

[SBB$^+$10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 77–91. Springer Verlag, 2010.

[SBD11] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, pages 43–56. ACM Press, 2011.

[SC75] J.M. Smith and P.Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM (CACM)*, 18(10):568–579, 1975.

[SC92] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the USENIX Conference*, pages 185–198. USENIX Association Berkeley, USA, 1992.

[Sch73] P.B. Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference (ACM)*, pages 106–113. ACM Press, 1973.

[Sch10] H. Schink. Sprachübergreifende Refactoring Feature Module. Master thesis (Diplomarbeit), University of Magdeburg, Germany, August 2010. [Available online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSchink.pdf`; accessed: July 16,2011].

[SD10] I. Schaefer and F. Damiani. Pure delta-oriented programming. In *Proceedings of the Workshop on Feature-Oriented Software Development (FOSD)*, pages 49–56. ACM Press, 2010.

[SdML04] G. Sittampalam, O. de Moor, and K.F. Larsen. Incremental execution of transformation specifications. *ACM SIGPLAN Notices*, 39(1):26–38, 2004.

[SH99] G. Saake and A. Heuer. *Datenbanken: Implementierungstechniken.* MITP-Verlag, 1999.

[SK10] H. Schink and M. Kuhlemann. Hurdles in refactoring multi-language programs. Technical Report FIN-007-2010, University of Magdeburg, Germany, 2010.

[SKAP10] N. Siegmund, M. Kuhlemann, S. Apel, and M. Pukall. Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 115–122. University of Duisburg-Essen, Germany, 2010.

[SKL06] D. Strein, H. Kratz, and W. Löwe. Cross-language program analysis and refactoring. In *Proceedings of the Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 207–216. IEEE Computer Society, 2006.

[SKR$^+$08] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, and G. Saake. Integrated product line model for semi-automated product derivation using non-functional properties. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 25–23. University of Duisburg-Essen, Germany, 2008.

[SKS$^+$08] S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, and G. Saake. Generating highly customizable SQL parsers. In *Proceedings of the Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 29–34. ACM Press, 2008.

[SKSL11] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel. Hurdles in multi-language refactoring of Hibernate applications. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, pages 129–134. SciTePress, 2011.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 268–285. ACM Press, 1996.

[SMC74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Smi85] B.C. Smith. The limits of correctness. *ACM SIGCAS Computers and Society*, 14,15:18–26, 1985.

[Smi90] D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering (TSE)*, 16(9):1024–1043, 1990.

[Smi91] D.R. Smith. KIDS: A knowledge-based software development system. In *Automating Software Design*, pages 483–514. AAAI/MIT Press, 1991.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45. ACM Press, 1986.

[Sof08] Software Systems Generator Research Group. *The jampack composition tool*, 2008. AHEAD tool suite v2008.07.22, manual [Available online: `http://www.cs.utexas.edu/users/schwartz/ATS/fopdocs/JamPack.html`, accessed: July 14,2011].

[SPTJ01] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML models. In *Proceedings of the Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML)*, pages 134–148. Springer Verlag, 2001.

[SR02] K.C. Sekaraiah and D.J. Ram. Object schizophrenia problem in modeling Is-Role-Of inheritance. In *Proceedings of the Inheritance Workshop*, pages 88–94, 2002. [Available online: `http://www.cs.jyu.fi/~sakkinen/inhws/papers/Sekharaiah.pdf`; accessed: July 18,2011].

[ŞR07] I. Şavga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 175–184. ACM Press, 2007.

[Sri92] A. Srivastava. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems*, 1(4):355–364, 1992.

[SRK$^+$08] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring non-functional properties in software product lines for product derivation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE Computer Society, 2008.

[SSS07] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in software product lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 167–173. IEEE Computer Society, 2007.

[Str91] B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1991.

[Stu10] M. Sturm. Debugging Generierter Software nach Anwendung von Refactorings. Master thesis (Diplomarbeit), University of Magdeburg, Germany, March 2010. [Available online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSturm.pdf`; accessed: July 16,2011].

[Sun03] Sun Microsystems, Inc. *JSR-000045 Debugging support for other languages 1.0 FR*, 2003. [Available online: `http://jcp.org/aboutJava/communityprocess/final/jsr045/index.html`; accessed: July 14,2011].

[Sun06] Sun Microsystems, Inc. *JSR-000221 JDBC^{TM}4.0 Specification Final Release*, 2006. [Available online: `http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html`; accessed: July 14,2011].

[SVEd09] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping stones over the refactoring rubicon. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 369–393. Springer Verlag, 2009.

[TB01] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.

[TBD06] S. Trujillo, D. Batory, and O. Díaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 191–200. ACM Press, 2006.

[TBD07] S. Trujillo, D. Batory, and O. Díaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 44–53. IEEE Computer Society, 2007.

[TBKC07] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.

[TC98] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. In *Proceedings of the Workshop on Reflective Programming in C++ and Java*, pages 56–60. University of Tsukuba, Japan, 1998.

[TCKI00] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Proceedings of the Workshop on Reflection and Software Engineering*, pages 117–133. Springer Verlag, 2000.

[TDDN00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE)*, pages 154–164. IEEE Computer Society, 2000.

[Tho05] D.A. Thomas. Refactoring as meta programming. *Journal of Object Technology (JOT)*, 4(1):7–12, 2005.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119. ACM Press, 1999.

[Tor04] M. Torgersen. The expression problem revisited. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 123–143. Springer Verlag, 2004.

[TSKA11] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proceedings of the Workshop on Variability-intensive Systems Testing, Validation & Verification (VAST)*, pages 270–277. IEEE Computer Society, 2011.

[TSSPL09] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86. ACM Press, 2009.

[TTS+08] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 37–52. ACM Press, 2008.

[vDD94] A. van Deursen and T.B. Dinesh. Origin tracking for higher-order term rewriting systems. In *Proceedings of the Workshop on Higher-Order Algebra, Logic, and Term Rewriting (HOA)*, pages 76–95. Springer Verlag, 1994.

[vDKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545, 1993.

[vdLSR07] F.J. van der Linden, K. Schmid, and E. Rommes. *Software product lines in action: The best industrial practice in product line engineering.* Springer Verlag, 2007.

[vdS04] T. van der Storm. Variability and component composition. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 86–100. Springer Verlag, 2004.

[VEd06] 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. ACM Press, 2006.

[Vit03] M. Vittek. Refactoring browser with preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE Computer Society, 2003.

[VRB00] J. Viega, P. Reynolds, and R. Behrends. Automating delegation in class-based languages. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 171–182. IEEE Computer Society, 2000.

[VRFS08] H. Venturini, F. Riss, J.-C. Fernandez, and M. Santana. A fully-non-transparent approach to the code location problem. In *Proceedings of the Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 61–68. ACM Press, 2008.

[VS10] L. VanderHart and S. Sierra. *Practical Clojure.* Apress, 2010.

[VV08] S. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *Proceedings of the Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 630–644. Springer Verlag, 2008.

[WB95] M.P. Ward and K.H. Bennett. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 5:25–47, 1995.

[WDS09] J. White, B. Dougherty, and D.C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software (JSS)*, 82(8):1268–1284, 2009.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[WGM08] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Software–Practice and Experience (SP&E)*, 38(10):1073–1103, 2008.

[Wid07] T. Widmer. Unleashing the power of refactoring. *Eclipse Corner Articles*, 2007. [Available online: `http://www.eclipse.org/articles/printable.-php?file=Article-Unleashing-the-Power-of-Refactoring/index.html`; accessed: July 14,2011].

[WLQ09] J.J. Willcock, A. Lumsdaine, and D.J. Quinlan. Reusable, generic program analyses and transformations. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 5–14. ACM Press, 2009.

[Woo97] B. Woolf. The Null Object pattern. In *Proceedings of the Conference on Pattern Languages of Program Design (PLOPD)*, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., 1997.

[WS91] D. Whitfield and M.L. Soffa. Automatic generation of global optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 120–129. ACM Press, 1991.

[WS97] D.L. Whitfield and M.L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.

[WSWN07] J. White, D.C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 129–140. IEEE Computer Society, 2007.

[WT09] S. Wehr and P. Thiemann. JavaGI in the battlefield: Practical experience with generalized interfaces. In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE)*, pages 65–74. ACM Press, 2009.

[XMEH04] B. Xin, S. McDirmid, E. Eide, and W.C. Hsieh. A comparison of Jiazzi and AspectJ for feature-wise decomposition. Technical Report UUCS-04-001, University of Utah, USA, 2004.

[Zel83] P.T. Zellweger. An interactive high-level debugger for control-flow optimized programs (summary). In *Software Engineering Symposium on High-Level Debugging*, pages 159–171. ACM Press, 1983.

[ZGJ05] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards Just-In-Time middleware architectures. In *Proceedings of the Conference on Aspect-Oriented Software Development*, pages 63–74. ACM Press, 2005.

[Zim95] W. Zimmer. Relationships between design patterns. In *Proceedings of the Conference on Pattern Languages of Program Design (PLOPD)*, pages 345–364. ACM Press/Addison-Wesley Publishing Co., 1995.

[ZJ03] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139. ACM Press, 2003.

[ZJ04] C. Zhang and H.-A. Jacobsen. Resolving feature convolution in middleware systems. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205. ACM Press, 2004.

[ZJ05] H. Zhang and S. Jarzabek. A bayesian network approach to rational architectural design. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 15(4):695–718, 2005.

[ZJY03] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 681–695. Springer Verlag, 2003.