Nr.: FIN-005-2010

Algebraic and Cost-based Optimization of Refactoring Sequences

Martin Kuhlemann, Liang Liang und Gunter Saake

Arbeitsgruppe Datenbanken

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-005-2010

Algebraic and Cost-based Optimization of Refactoring Sequences

Martin Kuhlemann, Liang Liang und Gunter Saake

Arbeitsgruppe Datenbanken

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

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

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

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

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

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 26.03.2010

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

Algebraic and Cost-based Optimization of Refactoring Sequences*

Martin Kuhlemann[1], Liang Liang[2], and Gunter Saake[3]

[1] University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de
[2] University of Magdeburg, Germany
leon.liangliang@hotmail.com
[3] University of Magdeburg, Germany
saake@iti.cs.uni-magdeburg.de

Abstract. Software product lines comprise techniques to tailor a program by selecting features. Selected features translate into sequenced program transformations which extend a base program. However, a sequence translated from the user selection can be inefficient to execute. In this paper, we show how we optimize sequences of refactoring transformations to reduce the composition time for product line programs.

1 Introduction

A feature is a characteristic of a program which is of interest to a user [12]. *Software product lines (SPLs)* comprise techniques to tailor the set of features of a program to user needs [16]. One technique to implement an SPL is to define code transformations which successively apply to a base program and add the desired program characteristic to it. These transformations can include aspects [30], refinements [4], refactorings [17], and others.

In SPLs, feature-adding code transformations are abstract operations which a user selects without knowing their implementation. As a result the user (unknowingly) may select transformations that undo each other in the sequence of transformation application. Such a non-optimal *refactoring plan* may be selected by accident (as the selector does not know the transformations) but may also be meaningful to reuse transformations.[1] While the composition result is correct and the composition process succeeds, the composition process is more expensive than necessary.

In this paper, we lean on database optimization techniques and optimize sequences of refactorings translated from a user selection of features. We discuss

* This paper summarizes and extends the Master's Thesis of Liang Liang [19].

[1] Suppose, in one configuration of an SPL two classes List and ArrayList should switch names then one of them must be renamed twice, e.g., List $\mapsto$ TestList $\mapsto$ ArrayList. In a second configuration, in which only List exists, the developer may wish to rename List into ArrayList, too, and for that both prior refactorings get *reused*. The second undoes the first refactoring but both are meaningful.

the theoretical basics as well as our prototype. Finally, we report on a number of case studies. In these case studies we show that with our prototype we could reduce composition time by up to 81%.

2 Background

In this section we introduce the concepts of refactoring along with feature-oriented programming and refactoring feature modules. These transformations are issue to optimization later.

2.1 Refactorings

Refactorings are code transformations that alter the structure of code but do not alter its functionality [24]. As refactoring descriptions like Rename Class are templates, a developer has to provide parameters to these templates to make them executable [23]. For example, to execute a Rename Class refactoring, the developer has to provide two parameters: the class to rename and the new class name. In common IDEs like Eclipse[2], the user provides such parameters by selecting code and answering GUI forms.

When a refactoring is parameterized and executed, the refactoring engine commonly executes two phases. First in the *verification phase*, preconditions are checked in the code to refactor to ensure the transformation to be performed does succeed, does not create an incompilable result, and does not alter functionality of the program. For Rename Class refactoring, the refactoring engine will check whether (a) the class to rename does exist and (b) the class created by the refactoring does not exist [26].

Second in the *transformation phase*, transformation actions are performed on the code elements specified as parameters for the refactoring. That is, for Rename Class, the specified class is renamed, constructors of the class are renamed, and finally every reference to the class or constructors is updated in the remaining code [8]. In the following, we denote a refactoring R that replaces a code element X with a code element Y by $R_{\mathsf{X}\mapsto\mathsf{Y}}$.

2.2 Feature-oriented Programming and Refactoring Feature Modules

Features are user-visible program characteristics of an SPL and are organized in feature models [12]. Features are implemented by code transformations in feature-oriented programming defined in feature modules [4]. The feature modules, however, are hidden from the user – she configures the SPL by selecting the feature modules based on their semantic description. Commonly, the feature modules add members and classes to a program, and extend methods (we call them *common features*). Recently, however, we discovered that structure of software also is a program characteristic which a user might be interested in [17].

[2] http://www.eclipse.org/

Restructuring transformations were added as SPL transformations to integrate programs, foster reuse, and to tailor non-functional properties of programs [17,27]. Feature modules which host such restructuring transformations were called *Refactoring Feature Modules (RFMs)* [17]. For a user, RFMs and common features are indistinguishable. When a user selects common features, code is added to the configured program such that the program provides certain functionality [4]. When a user selects RFMs the structure of the synthesized program is altered, e.g., classes are named differently than defined in the class-adding common feature.

In our running example for this paper in Figure 1, there is one common feature module *F1*. Additionally, there is a number of RFMs, *R2* to *R6*. When a user selects feature *F1* and does not select any RFM, the composed program will be a copy of the code of *F1*. When a user selects all features (top-down order), *F1* along with *R2* to *R6* the composed program will expose the functionality of *F1* but will have a different structure. Specifically, when all features are selected, then the resulting code will be a class myQueue with no members and a class Queue with a field _elements and a method pop.

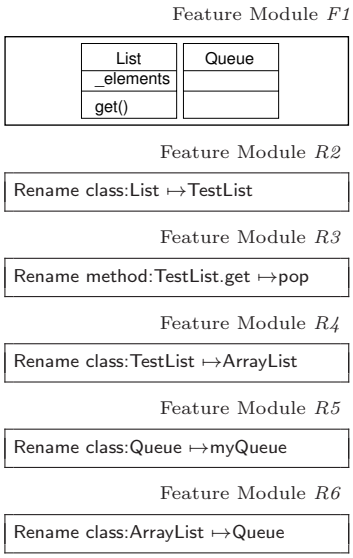
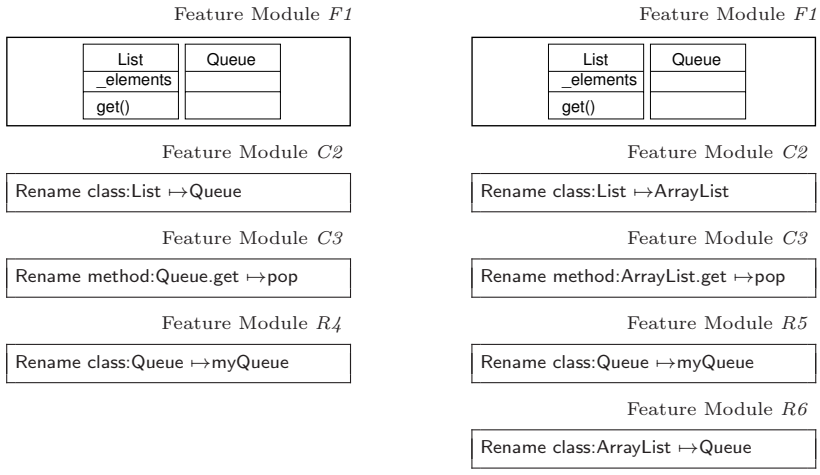


Fig. 1. Running RFM example.

3 Optimizing Refactoring Sequences

We consider two ways to optimize a given sequence of refactorings: optimizing the verification phases and optimizing the transformation phases of the sequenced refactorings. Optimizing *verification phases* in a sequence of refactorings means to check whether preceding refactorings establish preconditions of later refactorings. When a refactoring's precondition is satisfied by an earlier refactoring in a sequence, the program does not have to be validated for the latter refactoring, and thus not parsed and traversed *for verification issues* [26,14]. Thereby, checking a program might be *expensive* as the program to check might be large [14].

Optimizing the *transformation phase* for a sequence of refactorings means to fuse actions performed by successive refactorings. For example, we can fuse two successive refactorings if both refactorings rename the same method, i.e., we can replace two refactorings $R1_{\mathsf{A}\mapsto\mathsf{B}}$ and $R2_{\mathsf{B}\mapsto\mathsf{C}}$ by $Cx_{\mathsf{A}\mapsto\mathsf{C}}$. As we do not have to traverse the code twice to (parse it and set up the type system and) look for calls to the method and update them, we can gain performance benefits. The optimizations we will discuss work without and with prior code analysis, i.e., they work algebraic and cost-based respectively.



(a) Incorrect optimization result. (b) Correct optimization result.

Fig. 2.

3.1 Algebraic Optimization

In this work, we concentrate on fusing transformation phases of refactorings to improve composition performance. To optimize a given sequence of refactorings, we reorder sequenced refactorings and fuse them finally. We reorder refactorings to group refactorings of which action phases could be fused, i.e., where the output code element of the earlier refactoring is the input code element of the following refactoring. We identify these refactorings by analyzing the parameters of the sequenced RFMs. The reordered RFM sequence then is folded by fusing successive RFMs using fusing rules.

Basic Concept. The composition of our running example in Figure 1 can be optimized when all features contribute to a program. The class List gets renamed three times. We could reorder the figure's refactorings to first apply all refactorings which transform the initial class List (RFMs *R2*, *R4*, and *R6*), then the refactoring on method get (*R3*), and finally *R5* which transforms class Queue. After reordering, we could fuse $R2_{\mathsf{List}\mapsto\mathsf{TestList}}$ with $R4_{\mathsf{TestList}\mapsto\mathsf{ArrayList}}$, and $R6_{\mathsf{ArrayList}\mapsto\mathsf{Queue}}$ to a new Rename Class refactoring $C2_{\mathsf{List}\mapsto\mathsf{Queue}}$ as shown in Figure 2a.

Since we reorder refactorings, we may have to update parameters of commuted refactorings. In Figure 1, we have to update commuted *R3* to accept the parameter Queue.get instead of TestList.get (see feature *C3* in Fig. 2a) because *R4* and *R6* got reordered and precede *R3* finally.

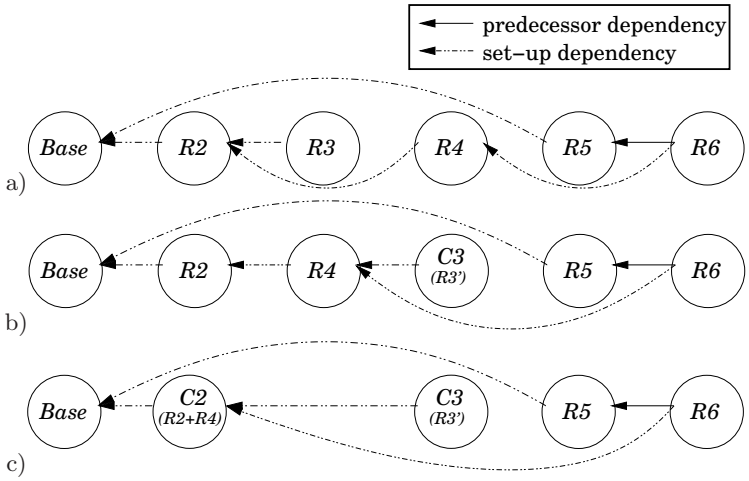


Fig. 3. Optimization steps in running example of Fig. 1.

The optimization result of Figure 2a is in error because newly created *C2* will create a second class Queue and thus will fail.[3] To prevent errors caused by reordering we have to analyze preconditions of refactorings in a refactoring sequence. For that, we – before reordering – analyze the sequenced refactorings for interdependencies. Especially, we look for two kinds of interdependencies: (1) *set-up dependencies* toward preceding RFMs where one preceding refactoring sets up some code elements required by a subsequent refactoring, and (2) *predecessor dependencies* toward preceding refactorings where a preceding refactoring requires another refactoring to establish a required deletion.[4] For our running example, we find that *R6* exposes a set-up dependency towards *R4* but also a predecessor dependency towards *R5*.[5] Furthermore, we find that *R3* and *R4* expose a set-up dependency each towards *R2* but no predecessor dependency towards any other refactoring.[6] The complete dependency graph for Figure 1 is given in Figure 3a.

To optimize the RFM sequence of Figure 3a, we iterate the sequence of refactorings and calculate potential fuse partners. For instance, we calculate, that *R2* could be fused with *R4* and *R4* with *R6* because they rename the same initial code element List. Using the computed dependency graph we *try* to reorder *R4* and *R6* according to their fusing potential. However, we only commute refac-

[3] In Java and alike languages fully qualified names, e.g., of classes, must be unique [9, p.123ff].

[4] A special predecessor dependency occurs when a Move Method RFM or Inline Method RFM follows an Extract Interface RFM and both operate the same class.

[5] *R4* creates class ArrayList which *R6* requires to exist. *R5* removes Queue which *R6* requires to not exist.

[6] *R2* creates class TestList which is required by *R3* and *R4*.

torings that do not have predecessor dependencies among each other. Further, we update the parameters of the two refactorings we commute when both expose set-up dependencies toward the same predecessor refactoring and share fully qualified names. For instance, we commute *R3* with *R4* because *R4* can potentially be fused with *R2*. As *R3* and *R4* both expose set-up dependencies toward *R2* and parameters share the identifier TestList, we update *R3* to become $C3_{\mathsf{ArrayList.get}\mapsto\mathsf{pop}}$, cf. Figure 3b. However, we do not reorder *R6* because its predecessor dependency towards *R5* disallows commuting with *R5*.

Fusing refactorings. In the second step of our optimization, we iterate the reordered and adapted list of RFMs and fuse successive RFMs when the fuse result again is a standard refactoring according to [8]. We fuse two refactorings when there is a set-up dependency between them, the complete precondition of the later refactoring is satisfied by the former refactoring, and when the fused refactoring again is a standard refactoring. This holds true, for example, when two Rename Method refactorings follow each other with $R1_{\mathsf{Stack.push}\mapsto\mathsf{add}}$ and $R2_{\mathsf{Stack.add}\mapsto\mathsf{insert}}$ – the fused refactoring again is a Rename Method refactoring $C1_{\mathsf{Stack.push}\mapsto\mathsf{insert}}$. We summarize fusing rules for refactoring actions in Table 1.

In our running example, we fuse the Rename Class RFM $R2_{\mathsf{List}\mapsto\mathsf{TestList}}$ with its successor Rename Class RFM $R4_{\mathsf{TestList}\mapsto\mathsf{ArrayList}}$ to become the new Rename Class RFM $C2_{\mathsf{List}\mapsto\mathsf{ArrayList}}$, see Figure 3c. The optimization result which corresponds to Figure 3c is shown in Figure 2b. Note, that we do not change *C3* and *R5* as they do not have optimization potential.

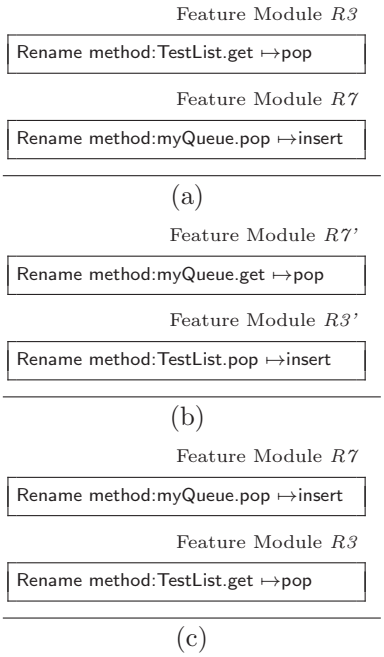


Fig. 4. Unknown commutativity.

Name capture. When a method A is renamed by a Rename Method refactoring, all methods that override A or that are overridden by A are renamed accordingly [8]. *Name capture* is an error in refactoring that occurs when methods override each other after a refactoring executed which did not override each other before the refactoring executed [24,23,29]. When reordering refactorings, we must guarantee that we do not introduce name capture, i.e., that the optimized refactoring sequence still produces *the same* program. For illustration, consider the RFMs in Figure 4a. By solely analyzing the RFMs we cannot decide whether myQueue.pop (required by *R7*) is created by *R3*, i.e., whether there is

Table 1. Fusing rules to optimize RFM sequences.

Preceding RFM	**Following RFM**	**Merged RFM**
Rename $Class_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C3}$	Rename $Class_{C1 \Rightarrow C3}$
Rename $Field_{F1 \Rightarrow F2}$	Rename $Field_{F2 \Rightarrow F3}$	Rename $Field_{F1 \Rightarrow F3}$
Rename $Method_{M1 \Rightarrow M2}$	Rename $Method_{M2 \Rightarrow M3}$	Rename $Method_{M1 \Rightarrow M3}$
Extract $Interface_{C1 \Rightarrow I2}$	Rename $Class_{I2 \Rightarrow I3}$	Extract $Interface_{C1 \Rightarrow I3}$
Rename $Method_{M1 \Rightarrow M2}$	Inline $Method_{M2}$	Inline $Method_{M1}$
Move $Class_{C1 \Rightarrow C2}$	Move $Class_{C2 \Rightarrow C3}$	Move $Class_{C1 \Rightarrow C3}$
Rename $Class_{C1 \Rightarrow C2}$	Collapse $hierarchy_{(C2,C3) \Rightarrow C3}$	Collapse $Hierarchy_{(C1,C3) \Rightarrow C3}$
Extract $Class_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C3}$	Extract $Class_{C1 \Rightarrow C3}$
Extract $Method_{M1 \Rightarrow M2}$	Rename $Method_{M2 \Rightarrow M3}$	Extract $Method_{M1 \Rightarrow M3}$
Extract $Class_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C3}$	Extract $Class_{C1 \Rightarrow C3}$
Extract $Class_{C1 \Rightarrow C2}$	Move $Class_{C2 \Rightarrow C3}$	Extract $Class_{C1 \Rightarrow C3}$
Extract $SC_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C3}$	Extract $SC_{C1 \Rightarrow C3}$
Extract $SC_{C1 \Rightarrow C2}$	Move $Class_{C2 \Rightarrow C3}$	Extract $SC_{C1 \Rightarrow C3}$
Extract $Superclass_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C3}$	Extract $Superclass_{C1 \Rightarrow C3}$
Extract $Superclass_{C1 \Rightarrow C2}$	Move $Class_{C2 \Rightarrow C3}$	Extract $Superclass_{C1 \Rightarrow C3}$
Push-Down $Field_{F1 \Rightarrow F2}$	Pull-Up $Field_{F2 \Rightarrow F1}$	∅
Push-Down $Method_{F1 \Rightarrow F2}$	Pull-Up $Method_{F2 \Rightarrow F1}$	∅
Rename $Class_{C1 \Rightarrow C2}$	Rename $Class_{C2 \Rightarrow C1}$	∅
Rename $Method_{M1 \Rightarrow M2}$	Rename $Method_{M2 \Rightarrow M1}$	∅
Rename $Field_{F1 \Rightarrow F2}$	Rename $Field_{F2 \Rightarrow F1}$	∅
Extract $Class_{C1 \Rightarrow C2}$	Collapse $hierarchy_{(C1,C2) \Rightarrow C2}$	∅
Extract $SC_{C1 \Rightarrow C2}$	Collapse $hierarchy_{(C1,C2) \Rightarrow C2}$	∅
Extract $Superclass_{C1 \Rightarrow C2}$	Collapse $hierarchy_{(C1,C2) \Rightarrow C2}$	∅
Extract $Interface_{C1 \Rightarrow I1}$	Collapse $hierarchy_{(C1,I1) \Rightarrow C1}$	∅
Rename $Method_{M1 \Rightarrow M2}$	Remove Setting $Method_{M2}$	Remove Setting $Method_{M1}$
Rename $Field_{F1 \Rightarrow F2}$	Inline $Temp_{F2}$	Inline $Temp_{F1}$
Introduce Explain. $Variable_{F1}$	Rename $Field_{F1 \Rightarrow F2}$	Introduce Explain. $Variable_{F2}$
Rename $Method_{M1 \Rightarrow M2}$	Encaps. $Collection_{M2 \Rightarrow \{M3,M4\}}$	Encaps. $Collection_{M1 \Rightarrow \{M3,M4\}}$
Introduce Foreign $Method_{M1}$	Rename $Method_{M1 \Rightarrow M2}$	Introduce Foreign $Method_{M2}$
Encaps. $Collection_{M1 \Rightarrow \{M2,M3\}}$	Rename $Method_{M2 \Rightarrow M4}$	Encaps. $Collection_{M1 \Rightarrow \{M4,M3\}}$
Repl. Param. with Explic. $Meth._{P1 \Rightarrow M1}$	Rename $Method_{M1 \Rightarrow M2}$	Repl. Param. with Explic. $Meth._{P1 \Rightarrow M2}$
Repl. Constr. with $FM_{M1 \Rightarrow \{M1,M2\}}$	Rename $Method_{M2 \Rightarrow M3}$	Repl. Constr. with $FM_{M1 \Rightarrow \{M1,M3\}}$
Introduce $PO_{M1 \Rightarrow \{M1,C1\}}$	Rename $Class_{C1 \Rightarrow C2}$	Introduce $PO_{M1 \Rightarrow \{M1,C2\}}$
Rename $Class_{C1 \Rightarrow C2}$	Inline $Class_{C2,C3 \Rightarrow C3}$	Inline $Class_{C1,C3 \Rightarrow C3}$
Rename $Class_{C1 \Rightarrow C2}$	Repl. SC with $Field_{C2,C3 \Rightarrow F1}$	Repl. SC with $Field_{C1,C3 \Rightarrow F1}$
Introduce Local $Extens._{C1 \Rightarrow \{C1,C2\}}$	Rename $Class_{C2 \Rightarrow C3}$	Introduce Local $Extens._{C1 \Rightarrow \{C1,C3\}}$
Repl. Array with $Object_{F1 \Rightarrow C1}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. Array with $Object_{F1 \Rightarrow C2}$
Dupl. Observed $Data_{C1 \Rightarrow \{C1,C2\}}$	Rename $Class_{C2 \Rightarrow C3}$	Dupl. Observed $Data_{C1 \Rightarrow \{C1,C2\}}$
Repl. Temp with $Query_{F1 \Rightarrow M1}$	Rename $Method_{M1 \Rightarrow M2}$	Repl. Temp with $Query_{F1 \Rightarrow M2}$
Repl. Method with $MO_{M1 \Rightarrow \{M1,C1\}}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. Method with $MO_{M1 \Rightarrow \{M1,C2\}}$
Repl. DV with $Object_{F1 \Rightarrow \{F2,C1\}}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. DV with $Object_{F1 \Rightarrow \{F2,C2\}}$
Repl. TC with $Class_{F1 \Rightarrow \{F2,C1\}}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. TC with $Class_{F1 \Rightarrow \{F2,C2\}}$
Repl. TC with $StrategyC_{F1 \Rightarrow C1}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. TC with $StrategyC_{F1 \Rightarrow C2}$
Repl. TC with $Strategy_{F1 \Rightarrow C1}$	Rename $Class_{C1 \Rightarrow C2}$	Repl. TC with $Strategy_{F1 \Rightarrow C2}$
Repl. Magic Number with SCo_{F1}	Rename $Field_{F1 \Rightarrow F2}$	Repl. Magic Number with SCo_{F2}

SC=Subclass;TC=Type Code;PO=Parameter Object;MO=Method Object;FM=Factory Method;SCo=Symbolic Constant;DV=Data Value

a set-up dependency from *R7* towards *R3*. There is such set-up dependency if myQueue is a subclass or superclass of TestList (then *R3* creates myQueue.pop) – reordering *R3* and *R7* would then require to update the parameters of both refactorings (see Fig. 4b). If myQueue is not a subclass or superclass of TestList, then reordering both refactorings requires no update to their parameters (see Fig. 4c). Name capture must also be prevented for fields (field hiding [9, p.206] – similar to macro extension [15]).

We present three approaches which avoid name capture. In approach #1, we track which refactoring parameter (fully qualified name) emerges out of which code element in the base code. By analyzing relationships between the code elements in the base code we can then decide whether to update the refactoring parameters or not. In approach #2, we disallow reordering of two refactorings when both reference methods, e.g., Rename Method refactoring, or when both reference fields. However, we only must disallow reordering when field or method names match in the refactorings to be reordered. In approach #3, we define all the elements, which a refactoring alters inside feature modules. As a result, we know all (overridden) methods which are effected by a Rename Method RFM. However, we do not consider the last approach practicable because methods that override a renamed method may change across configurations and we cannot define an RFM for every configuration.

Heuristical reordering. Reordering *itself* can produce performance benefits for the composition process, too. For example, when a Rename Field RFM follows an Encapsulate Field RFM[7], then reordering is beneficial though both RFMs cannot be fused. The reason is that the field to be renamed can be referenced multiple times in the transformed code but is only referenced twice after encapsulating it (inside the get and set method).

Secondly, to reorder a Hide Method refactoring[8] with a Rename Method refactoring is beneficial. After hiding the method, the composer can reason on the new visibility qualifier of that method and thus can prune the code traversed for renaming. For example, if hiding the method push produces a private method then for renaming the method the composer just needs to traverse the class (as no further references can exist). Similar optimizations are possible when a Rename Field refactoring follows a Hide Field refactoring.

Search spaces. We could create sets of optimized refactoring plans during algebraic optimization phase which all generate the same code. Doing so, we can find additional optimization potentials. For instance, at the moment we do not optimize the following sequence of refactorings because we cannot detect any optimization potential:

[7] Encapsulate field adds get and set methods for the field to encapsulate. Second the refactoring transformation replaces every reference to this field by a call to either the get or set method.

[8] Hide Method refactoring reduces the visibility of the method as far as possible [8].

$$R1_{RenameClass:\mathsf{C1}\mapsto\mathsf{C2}} \bullet R2_{MoveClass:\mathsf{C2}\mapsto\mathsf{C3}} \bullet R3_{RenameClass:\mathsf{C3}\mapsto\mathsf{C4}} \quad (1)$$

We can fuse neither *R1* with *R2* nor *R2* with *R3* because the resulting refactoring would not be a standard refactoring – fusing them would exceed our set of operations.[9] However, we also do not consider optimization potential between *R1* and *R3* because the output identifier of *R1* does not match the input identifier of *R3*. If we commute *R1* with *R2* or *R2* with *R3*, however, then a *new* optimization potential emerges between (reordered) *R1* and (reordered) *R3*.

3.2 Cost-based Optimization

We can analyze the code to be refactored to estimate the execution costs for individual refactorings; from there we can further optimize a refactoring sequence. We call optimizations which are based on code analyses and cost estimations *cost-based optimization*. We envision to identify refactorings which alter distinct parts of a program. If we can reorder these refactorings to succeed each other, we can *parallelize* their execution, i.e., we can load the distinct program parts in parallel. To implement that, we envision to collect visibility qualifiers and inheritance hierarchies from the program to refactor. If then the visibility of two code elements is very restricted, e.g., private or protected, and both occur in different class (hierarchies) according refactorings perform on distinct pieces of code.

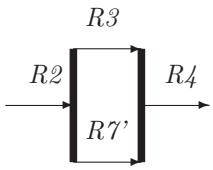


Fig. 5. Parallel RFM actions.

As an example, consider the Rename Method refactorings $R3_{\mathsf{TestList.get}\mapsto\mathsf{pop}}$ and $R7_{\mathsf{myQueue.pop}\mapsto\mathsf{insert}}$ where both methods are analyzed to be qualified as protected. TestList shall neither a superclass nor a subclass of myQueue and thus *R3* and *R7* transform distinct parts of a program. In that case we can infer an optimization potential and try to make both refactorings successors. We then can load TestList and myQueue in parallel and execute *R3* and *R7* in parallel as shown in Figure 5. We can also parallelize Rename Field and Rename Method RFMs if according fields or methods are qualified as private and all are hosted in different classes.[10]

If the visibility is private or protected and – in the latter case – the inheritance hierarchy is small, then we can reduce the code which must be loaded in order to refactor it. This reduces the number of buffer misses and thus increases performance.[11]

[9] We could provide composite refactorings which do renaming and moving within one step (as shown before [14]) but we refrained due to the infinit number of possible refactoring combinations [14].

[10] Name capture cannot occur for private elements in Java and alike languages [9, p.228].

[11] Buffer misses may occur when an inappropriate page replacement strategy is used by the operating system.

4 Case Studies

We now report on our prototype implementation and its evaluation.

4.1 Prototype

We implemented the presented approach for algebraic optimization of RFMs. Currently, a *separate* optimizer prototype operates RFMs in a step separately *before* the composer tool runs. It first tries to reorder RFMs if they expose optimization potential.[12] After that, the prototype fuses RFMs according to the rules presented before, cf. Tab. 1.[13] Finally, the prototype generates RFMs into a new folder *Optimized* and generates a new refactoring plan which uses the new RFMs. In future works, we think over integrating the optimizer tool into the composer tool. Integrating both tools will alleviate performance penalties in the current prototype of loading RFMs twice (once for optimization and once for composition) and of writing optimization result to folder *Optimized*.

4.2 Study Setup

To evaluate the proposed optimization approach, we now analyze the composition time of RFM-featured SPLs. We compare the times of composing the unoptimized sequence of common features and RFMs with the composition time of the generated optimized sequence.[14]

We took programs of different size and purpose as study objects. We composed the common feature modules and RFMs and took the composer's runtime. Then we run our optimizer tool and took its runtime, too.[15] The tool creates the folder *Optimized* together with the optimized sequence of RFMs. Finally, we compose the common feature modules and RFMs inside the *Optimized* folder and compare the composition time to the time of the unoptimized composition. We give an overview on our measurements in Table 2.[16]

In order to analyze the effect of a growing number of RFMs, we applied sequences of RFMs of different length to individual programs. In order to analyze the effect of a growing code size on the performance of RFM sequences (and thus

[12] Detecting name capture is not yet implemented.

[13] In our prototype, we implemented the first five fusing rules of Table 1.

[14] The optimizer prototype solely generates RFMs into the *Optimized* folder but does not copy common feature modules (it does copy RFMs). To measure the composition performance for the optimized RFM sequence, we manually copy the common feature modules into the *Optimized* folder.

[15] To estimate the potential of future integration with the composer tool we splitted the runtime of the optimizer into RFM loading time, optimization process time, RFM-writing time, and time to remove temp folders (clearing time).

[16] The measurements were performed on a Microsoft Windows XP Home Edition SP2 on an Intel® Core™2 CPU T5500 @ 1.66GHz, 667MHz FSB, 0.99 GB RAM. The given measurements are *averages* of 10 individual runs, Liang lists the single run times in [19].

Table 2. Measured tool run times (in *ms*).

Program	Refactorings	#SLOC*	Unopt. Comp.†	Loading RFMs	Optimization	Write RFMs&Plan	Clearing up	Opt. Comp.♭
Simple List (a)	1x Extract Interface, 1x Rename Method, 2x Rename Class, 1x Inline Method	19	12018.6	8834.4	9.4	40.6	50.2	9870.4
Simple List (b)	5x Rename Class, 3x Rename Method	19	12840.7	9271.9	9.5	55.8	64.2	9546.8
Simple List (c)	1x Rename Field, 4x Rename Class, 4x Rename Method, 1x Encapsulate Field	19	16359.3	8942	20.3	50	62.3	10412.3
TankWar	1x Rename Field, 4x Rename Class, 4x Rename Method, 1x Encapsulate Field	∼1K	31934.2	8078.1	18.8	39.2	70.2	14093.6
Workbench.texteditor (a)	1x Rename Field, 4x Rename Class, 4x Rename Method, 1x Encapsulate Field	∼16K	172162.4	18365.6	17.2	40.7	326.4	83561.1
Workbench.texteditor (b)	8x Rename Class, 9x Rename Field	∼16K	253831.2	18174.9	23.3	31.6	218.6	59731.2
Workbench.texteditor (c)	27x Rename Class, 28x Rename Field	∼16K	769617.5	75790.6	101.4	143.8	1596.9	61292.1
ZipMe	1x Rename Class, 2x Move Class	∼3K	20461	7718.9	10.8	39	98.4	20281.4

*lines of source code without RFMs; †unoptimized composition; ♭optimized composition

the optimization benefit) we measure small-scale cases to large-scale cases. As we do not change the composition of common features we prune the studies to only have one common feature module each.

Simple List. As a proof of concept we applied three different sequences of RFMs to a *conceptual* list implementation, the sequences are shown in Figure 6. According RFM sequences calculated by our prototype are given in Figure 7. In the studied case of Figure 6a, we fuse an RFM *R1* that extracts the interface AbstractList from class List with reordered RFMs *R3* and *R5* which rename the extracted interface (*C1* in Fig. 7a). In the sequence shown in Figure 6b, we detect the potential of fusing *R2*, *R5*, and *R8* but can only reorder and fuse *R5* with *R2* – *R8* cannot be fused because it cannot be commuted with *R4* (predecessor dependency). Reordering of *R5* requires the prototype to update the parameters of *R3* (*C2* in Fig. 7b).

TankWar. We analyzed TankWar an SPL of arcade games for desktop computer and handy developed prior to this evaluation at Magdeburg University. The study is still small-scale but provides functionality (in contrast to the Simple List case).

Workbench.texteditor. In order to analyze the performance effect of optimizing RFM sequences, we further must pay attention to the size of the program to be refactored. For that, we reused a large-scale study of the Eclipse[17] library workbench.texteditor from prior work [18]. To this library, we applied three different sequences of RFMs with a length ranging from 10 to 55 RFMs.

ZipMe. We finally analyzed a study of a compression library ZipMe from prior work [17] which showed us that our optimization effort may be worthless and, thus, derogatory. That is, in the ZipMe study, there is no optimization potential and thus, the runtime of our optimizer tool directly increases composition time.

In the Table 2, we summarize the measured runtimes of the optimizer tool as well as the runtimes of the composition tool on the unoptimized and on the optimized RFM sequence. In some cases we gained performance increases[18], e.g., for case Workbench.texteditor (c) we gained a performance benefit of 81% through optimization. In many cases, composition time increased with optimization. For example, the unoptimized composition time for case Simple List (a) is 12018.6ms and the optimized composition time including optimization time is 18805ms, i.e., a performance loss by 56%. Nevertheless, we did not fail optimizing. The increased composition time is caused by the optimizer prototype operating independently from the composer tool. Times for loading RFMs and writing optimization results, thus, contribute to both the composition tool *and* the optimizer. When the optimizer is integrated with the composition tool (possible future work), RFMs would be loaded only once and the need to write the

[17] http://www.eclipse.org/

[18] $Unoptimized\ composition\ time > (Loading\ Time + Optimization + Write\ RFMs\&Plan + Clearing\ time + Optimized\ composition\ time)$

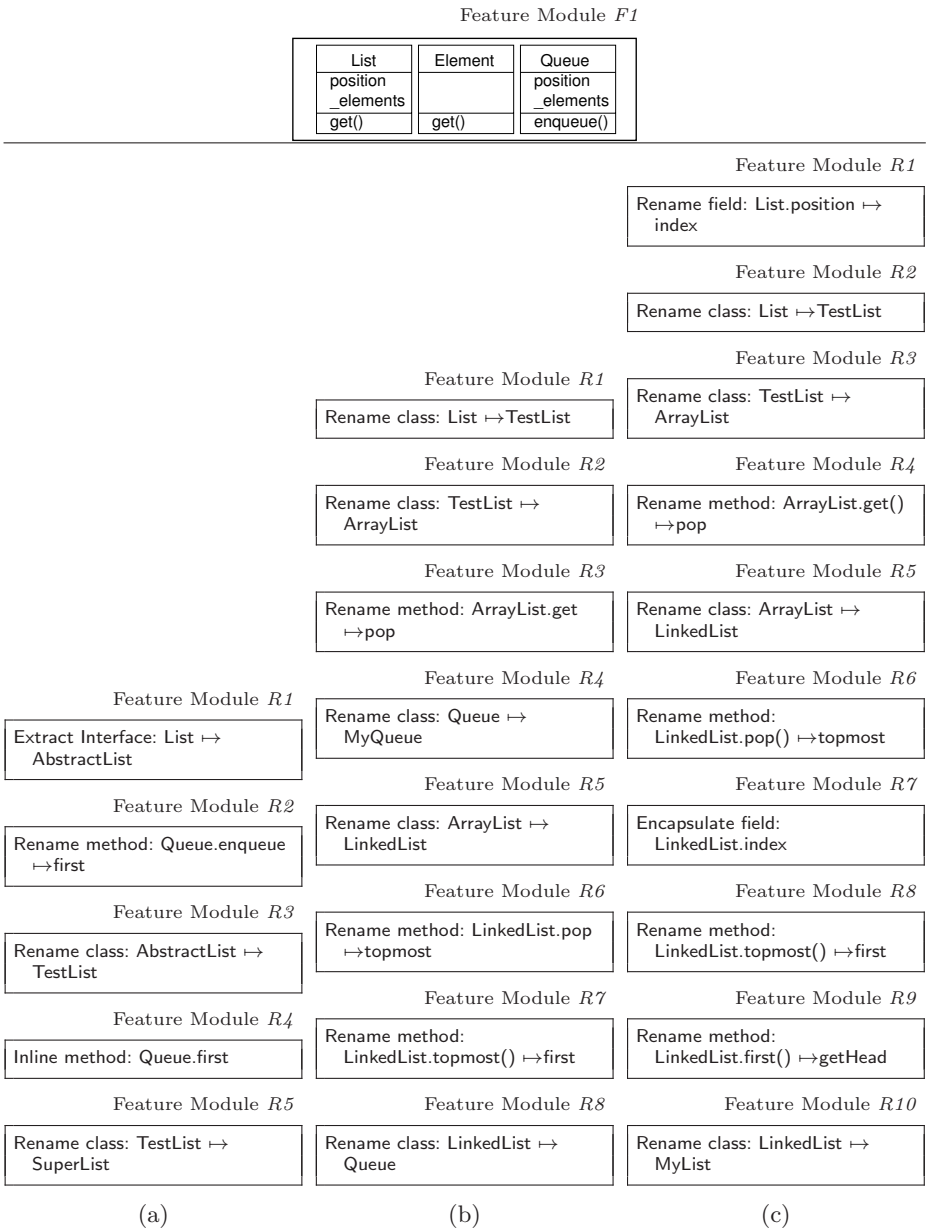


Fig. 6. Simple list study.

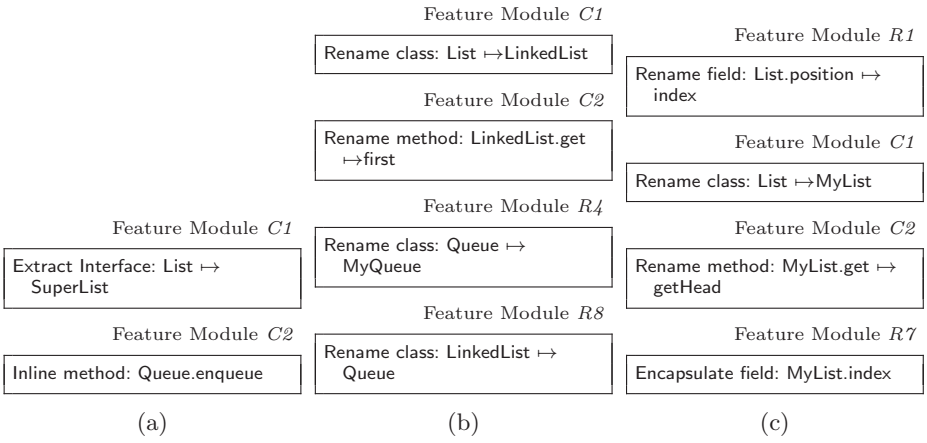


Fig. 7. Simple list study optimization result.

optimized RFM sequence to harddisk vanishes. To respect this, we split the runtime of the optimizer and separated times for loading the RFMs and writing the RFMs from the actual optimizing process. That is, in future work, RFMs are considered to be loaded once in the composer, optimized and executed by the composer, without reloading them and without writing the optimized sequence. When neglecting the costs of loading RFMs twice and writing optimized RFM sequences, we get a significant performance benefit for all cases but the ZipMe case (no fusing rules were applicable for the ZipMe case).

From the measurements we observed that the optimization benefit increases with a growing size of the program to be transformed sequentially, the biggest performance benefits were measured for the biggest program (Workbench.texteditor). We also observed that with a growing number of RFMs with optimization potential the optimization benefit increases, too. In the case of ZipMe, the optimizer could not produce a benefit and, thus, for this case optimization effort is derogatory.

4.3 Threats to Validity

The measurements and benefits are specific in two respects. First, they depend on the performance of loading RFMs. If to load an RFM takes a long time, reducing the number of loads saves a lot time. Second, the measurements and benefits depend on the time of executing a single RFM action. If executing a single RFM action takes a long time, reducing the number of executions saves a lot time.

The RFM composer tool we used (the only one we know of, downloaded Feb 2^{nd}, 2010) is written for flexibility and not for performance. Thus, for other RFM composers the numbers may be different. Nevertheless, we expect for those tools performance benefits as well when optimizing RFM sequences.

5 Related Work

Researchers composed transformations and refactorings to composite transformations (refactorings) before, e.g., [26,14,6,13]. They intended to group refactorings or to guarantee applicability or to improve refactoring execution time of the composite refactoring. Others formalized the refactoring effects and also analyzed preconditions of individual refactorings [23]. We focus on the performance gained through transforming a sequence of refactorings. For that, we *reorder and replace* single refactorings as well as (sub)sequences of refactorings and create a different ad-hoc sequence of refactorings. We concentrate on optimized refactoring sequences that only consist of standard refactorings because with RFMs refactorings are operations to which optimized sequences are limited to, i.e., we stay in the space of standard refactoring operations.

Dig fuses sequences of refactorings [7, p.95], sequences which were recorded independently. He adapts the parameters of refactorings in order to sequentialize the according refactorings. Similarly, Lynagh fuses patches and for that resolves conflicts by commuting and reverting patches [20]. It may happen that by fusing sequences of patches and refactorings, the resulting sequence may shrink. In contrast to prior work we *intend* to shrink a *single* sequence and for that fuse refactorings and reorder them.

Researchers describe how to calculate dependencies between transformations in general and refactorings in particular [21,22]. We also compute dependencies between refactorings, so prior research can be seen as a basis for our research. Based on dependencies between refactorings, we introduce fusing rules for individual refactoring actions and allow to update refactoring definitions (update refactoring parameters) when reordering. Further, we analyze optimizations of refactoring sequences based on analyses of the code to refactor (cost-based optimization).

Pérez uses artificial intelligence techniques to derive refactoring plans that minimize code smells [25]. We transform sequences of refactorings using fusing rules in order to yield performance benefits for their executions. Summarizing, we aim at different things and, thus, probably produce different refactoring sequences.

Relational algebra organizes a set of operations (Selection, Projection, Join) users can execute on databases [5,10]. With SQL, a user of database management systems, however, commonly describes *declaratively* the information she needs [28]. The algebra expression translated from the declarative query may be suboptimal and thus it is optimized *algebraically* and *cost-based* [11]. Algebraic optimization applies rules to the operation plan without analyzing the data, e.g., selections are reordered to execute early and projections are fused [10]. Cost-based optimization uses meta data of the database relations to fasten the query even more, e.g., whether relations are sorted [28]. In distributed database management systems, the query result can be computed on different systems in parallel to improve query time [1,5,11]. With features, a user declaratively defines the program she needs but does not formulate their implementations. During composition, the features are translated into sequenced common feature mod-

ules and RFMs – a sequence which may be suboptimal. In this paper, we showed how a sequence of refactorings inside RFMs can be optimized algebraically and cost-based, i.e., with and without analyzing the code to refactor. In our envisioned cost-based optimization we parallelize RFMs to improve composition time which then will closely correlate to parallel database management systems. Of course, database management systems do not execute program transformations.

Batory et al. related program transformations to category theory and, thus, sketched the formal basis of our optimizations [2,3]. Our fusion rules and heuristical reordering of refactoring transformations implement Batory's abstract concepts of combining transformation arrows. We additionally presented ideas on optimizing sequences of refactorings cost-based.

6 Conclusions

Product line users tailor programs by selecting features. Selected features translate into program transformations which execute sequentially on a base program. Thereby, a sequence translated directly from a user selection can be inefficient. In this paper, we showed how to optimize sequences of refactoring transformations to reduce the composition time of product line programs. We presented our prototype and evaluated the concept in several case studies. We observed that the optimization concept reduces the time to compose a program in most case studies.

Acknowledgements

The authors thank Don Batory and Andreas Lübcke for helpful discussions and for giving hints on earlier versions of this paper.

References

1. 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.
2. D. Batory. A modeling language for program design and synthesis. *Lecture Notes in Computer Science (LNCS)*, 5316:39–58, 2008.
3. D. Batory. Using modern mathematics as an fosd modeling language. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 35–44, 2008.
4. D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.
5. S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 34–43, 1998.
6. M. Ó Cinnéide and P. Nixon. Composite refactorings for java programs. In *Workshop on Formal Techniques for Java Programs (FTfJP)*, pages 129–135, 2000.
7. D. Dig. *Automated upgrading of component-based applications.* PhD thesis, University of Illinois at Urbana-Champaign, 2007.

8. M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification.* Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2005.
10. 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.
11. M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.
12. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
13. G. Kniesel. A logic foundation for program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, 2006.
14. G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.
15. E. Kohlbecker, D.P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the Conference on LISP and Functional Programming (LFP)*, pages 151–161, 1986.
16. C. W. Krueger. New methods in software product line practice. *Communications of the ACM (CACM)*, 49(12):37–40, 2006.
17. M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 106–115, 2009.
18. M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 177–186, 2009.
19. L. Liang. Optimizing sequences of refactorings. Master thesis, University of Magdeburg, Germany, MAR 2010.
20. I. Lynagh. An algebra of patches. http://urchin.earth.li/ ian/conflictors/paper-2006-10-30.pdf, 2006.
21. T. Mens, G. Kniesel, and O. Runge. Transformation dependency analysis - a comparison of two approaches. In *Actes des journéees Langages et Modèles à Objets (LMO)*, pages 167–184, 2006.
22. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.
23. T. Mens, N. v. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
24. W.F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
25. J. Pérez. Enabling refactoring with HTN planning to improve the design smells correction activity, 2008.
26. D.B. Roberts. *Practical analysis for refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.
27. 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 International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 115–122, 2010.
28. 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.

29. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.
30. C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.