Otto-von-Guericke University Magdeburg

School of Computer Science
Department of Technical and Business Information Systems

Master Thesis

Optimizing Sequences of Refactorings

Author:

Liang Liang

March 1, 2010

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Martin Kuhlemann

Otto-von-Guericke University Magdeburg
School of Computer Science
P.O.Box 4120, D–39016 Magdeburg
Germany

Liang, Liang:
Optimizing Sequences of Refactorings
Master Thesis, Otto-von-Guericke University
Magdeburg, 2010.

Contents

List of Figures

List of Tables

List of Abbreviations

SPL	Software Product Lines
FOP	Feature Oriented Programming
RefOp	Refactoring Operation
RF	Rename Field
RM	Rename Method
RC	Rename Class
EI	Extract Interface
IM	Inline Method
EF	Encapsulate Field
MM	Move Method
SA	Substitute Algorithm
CH	Collapse Hierarchy
AI	Artificial Intelligent

Chapter 1

Introduction

1.1 Motivation

Feature oriented programming (FOP), an extension of the paradigm for object oriented programming, can be used to implement software product lines in terms of features[CE00]. In the FOP, a feature is an increment in program functionality [BSR03]. Technically, the code of different classes associated to one feature is merged into one feature module. Assigning a feature to a configuration, which is a combination of features, causes the new feature module to superimpose the old feature module. That is, new members and classes are added or old classes are refined [BSR03]. In other words, the feature is implemented by the feature module. FOP can produce different concrete programs by selecting or deselecting features on a feature model[CE00][Bat05]. Therefore, feature oriented design improves reusability of classes by using feature modules to add new classes to a program, add new members, or extend members of existing classes.

Distinguished from the concept of FOP, refactoring is a disciplined process of changing a program in such a way that it does not alter the program's behavior while improves its existing members and classes. Refactoring defined by Martin Fowler is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing it observable behaviour"[Fow99]. Hence, a refactoring describes a process of transforming the structure of program while keeping its functionality. For instance, "rename method refactoring" changes the name of a method to reveal the method's purpose but does not change its functionality.

Refactoring feature modules (RFMs) combine refactorings and feature modules through encapsulating refactorings and sequences of refactorings in feature modules[KBA09]. A refactoring feature module can be seen as a special feature in the feature model[KBA09]. This kind of the special feature transforms the structure of program but does not add program functionality. Thus, both refactoring features and normal features can be selected in the feature model, no matter what the type of features is. Furthermore, the compiler can successively compose the selected features (normal feature or refactoring) according to the order in the feature model[KBA09]. By using this way, software product lines can be tailored regarding non-functional properties[SKAP10]. But the encapsulated refactorings in feature modules might be selected by user in a way which produces a disadvantageous sequence. For example, a method may be renamed twice. When the disadvantageous sequence of refactorings is applied to a program, it may cause a high compilation effort. With the number of refactorings growing, the high effort of compilation becomes unfeasible. In order to reduce the effort of compilation, disadvantageous sequences of refactorings should be optimized before compiled. This thesis focuses on optimizing sequences of refactorings to reduce the effort of compilation.

1.2 Goals

In this thesis, a theoretical framework about how to optimize disadvantageous sequences of refactorings to reduce compilation time when composing corresponding sequences of RFMs to a base feature-oriented program is described. As a concept demonstration, the presented concepts in a prototype that integrated to existing compose tool in four case studies are implemented and evaluated.

1.3 Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** describes the background to the thesis in detail, i.e., relevant concepts of refactorings, feature oriented programming and refactoring feature modules.
- **Chapter 3** firstly gives an introduction to the theoretical foundation of predefined refactorings in the RFMs and explores relationships among refactorings in given sequence. Moreover, we present the theoretical framework of the logical optimization to express the whole process of the optimization of one user-defined sequence of RFMs. Additionally, the concept of physical optimization to complement optimization in logical level is discussed.
- **Chapter 4** implements the presented concepts in a prototype of optimization tool.
- **Chapter 5** reports on four case studies and evaluates the compilation time of different sequences in these case studies by comparing the original user-defined sequences of RFMs with the optimized sequences of RFMs. In these case studies, the compilation time of composing user-defined sequences is reduced by up to 81.9% with our optimization tool.
- **Chapter 6** reports on the related work.
- **Chapter 7** concludes the thesis with a summary and presents ideas for future work.

Chapter 2

Background

In this chapter, the theoretical background that is necessary for understanding this thesis is provided. Since we mainly analyze sequences of refactorings in the rest of the thesis, we firstly explain the concept of refactorings and composition of refactorings. Subsequently, we briefly introduce the concept of feature oriented programming (FOP). Lastly, we provide the principle of refactoring feature modules (RFMs) in detail for understanding how to integrate our work into refactoring feature modules.

2.1 Refactoring

The application of refactoring in the process of software development has gained much attention in recent years[Fow99][Opd92][Rob99]. Especially, it becomes a important part of the movement towards agile processes[Coc06] and extreme programming[Bec99][BF01] by improving the design of existing code through the software lifecycle. In this section, we begin with an introduction of the concept of refactoring. Furthermore, the principle of composition of refactorings is introduced.

2.1.1 The Concept of Refactoring

The term refactoring was originally introduced in Opdyke's dissertation[Opd92]. He proposed refactoring as a disciplined technique with behavior-preserving program transformations in order to support the iterative design of object-oriented programs. Later, Fowler emphasizes that the intention of refactoring is to improve the design and introduces the definitions of refactoring as follows:

Definition 2.1 *"Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."([Fow99])*

Definition 2.2 *"Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior."([Fow99])*

To illustrate, Figure 2.1 presents an apparent example, "Rename Method" refactoring, which is described in book[Fow99]. If the name of a method does not reveal its purpose, the old name of the method to the new one in all calls that occur in the program should be changed. We execute “Rename Method” refactoring two times to the initial program which is shown in Figure 2.1(a). One transforms old method name “foo” to “enqueue”, the other transforms old method name “bar” to “dequeue”. As illustrated in this example, the transformed program improves the overall readability of the initial program.

```java
class Queue{
  private LinkedList elements = new LinkedList();

  public void foo(Element element){
    elements.add(element);
  }

  public Element bar(){
    elements.removeFirst(element);
  }
}
```

(a) Before refactoring

```java
class Queue{
  private LinkedList elements = new LinkedList();

  public void enqueue(Element element){
    elements.add(element);
  }
  public Element dequeue(){
    elements.removeFirst(element);
  }
}
```

(b) After refactoring

Figure 2.1: Rename Method Refactoring

Besides “Rename Method” refactoring, Fowler has already introduced a comprehensive catalogue of 68 different kinds of refactorings[Fow99] and described them in the following way: Firstly, he gives each refactoring a name and follows the name with a short summary that describes the refactoring. Secondly, he introduces a motivation that describes why the refactoring should be done. Thirdly, a mechanic is given to describe how to carry out this refactoring step-by-step. Finally, he shows a corresponding example to illustrate how this refactoring works. In practice, performing refactorings by hand usually relies on program complication and test cycles. For example, on one hand complication detects whether a new method has the same name as an already existing method for “Rename Method” refactoring. On the other hand, it is necessary to guarantee that the behavior of a program is not transformed after refactoring by testing. In summary, Opdyke and Fowler describe refactorings as combinations of preconditions whose purpose is to guarantee the execution of refactorings and actual transformation steps that are described through using natural language.

Due to the fact that performing refactorings manually is tedious and error-prone, Roberts developed an automatic refactoring tool to reduce program analysis by asserting preconditions and postconditions on the abstract syntax tree of a Smalltalk program to assist refactoring[Rob99]. The preconditions and postconditions are all described as first-order predicates, which allows the calculation of properties of relationships among refactorings. His work also contains certain principles of composition of refactorings. Based on the preconditions and postconditions, the preconditions for the chain of the composite refactorings can be derived. Ó Cinnéide et al. use the approach of Roberts in order to compose refactorings related to design patterns. Besides the composition of preconditions, he introduces the concept of composition of postconditions. We will briefly review the concept of the composition of refactorings.

2.1.2 The Composition of Refactorings

Due to the concept of composing refactorings is an important part of our approach, it is necessary that we examine relevant approaches for composition of refactorings from existing researches. The earlier work is from the Roberts. He introduces how to derive the precondition of a composite refactoring from the preconditions of the individual refactorings in a sequence of refactorings[Rob99]. When there are dependencies among refactorings in the sequence, the precondition of the composite refactorings does not simply combine the preconditions of the individual refactorings together. This is because results of earlier refactorings will establish satisfactions of some preconditions of the later refactorings.

The paper[CN00] that extends the approach of Roberts describes a concept of the composite refactorings for Java program. Distinguished from the approach of Roberts, the core concept is to compute what the overall preconditions and postconditions are for the whole chain of primitive refactorings. It means that the derivation of a postcondition for a composite refactorings is also considered. We review some detailed concepts of composing refactorings shown in this thesis, we have already known each refactoring has its own precondition and postcondition. The approach for composing this sequence is to evaluate the precondition and postcondition for each individual refactoring step by step in the given sequence. The precondition and the postcondition for the whole sequence by using the approach can be calculated. Theoretically, we should discuss how to calculate precondition and postcondition for a two-refactoring-sequence firstly. Then, we can derive the composite conditions for any length sequence by iteratively analyzing pair of refactorings.

To illustrate, we compose the two refactorings R_1 and R_2. We borrow the notion from the paper[CN00]. For a general refactoring R_i, its precondition and postcondition are denoted by $preR_i$ and $postR_i$ respectively. In Figure 2.2, we can compose the preconditions and postconditions for a two-refactorings-sequence. There are three potential

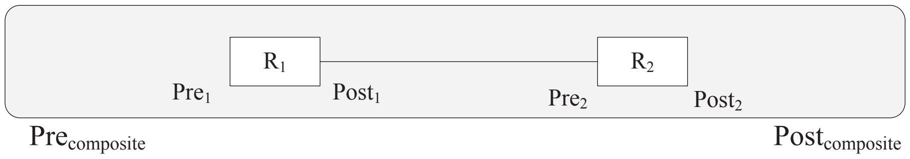


Figure 2.2: A Chain of Refactorings

benefits for the composition of the two refactorings:

- the precondition for R_1 may includes part of the precondition of R_2.

- $postR_1$ may guarantee $preR_2$. In other words, the execution of R_1 provides the satisfaction of part of precondition of R_2.
- if $postR_1 \wedge postR_2$ leads to contradiction, the part of $postR_1$ that causes the contradiction is dropped.

Essentially, the postcondition of the above concepts is similar to a abstract concept of transformation effect of refactorings. Therefore, in our optimization approach, the types of optimization potential, which are applied to pairs of refactorings, are divided to two categories: composition potential and simplification potential. Similar to the concept of composition, the composition potential can eliminate the precondition of a pair of refactorings and keep the two transformations. Distinguished from the the concept of composition, the simplification potential not only can eliminate the precondition of a pair of refactoring but also can compose both transformations to an updated transformation.

2.2 Feature Oriented Programming

FOP is one technique to implement software product lines and improve reusability of object-oriented classes[BSR03]. Distinguished from refactorings that restructure existing members and classes, FOP transforms a base program by creating new members and classes or extending them. In this section, we introduce the concept of features and feature oriented design respectively.

2.2.1 The Concept of Features

The intention of software product lines(SPL) is to generate tailored programs from different set of similar products distinguished by the features they implement[CE00]. In concept, a tailored program is composed from a base program that includes some base code shared by all other features[CE00]. A user can create a feature model by a domain requirement to select the corresponding features of the SPL. The feature model is essentially a configuration file that not only defines the selected features but also defines the order of composing selected features onto base program.

As one implementation of software product lines, the basic concept of FOP is feature. Batory et al. define *feature* as a "product characteristic that is used in distinguishing programs within a family of related programs."[BSR03]. In FOP, features correspond to functionalities that programs provided and a single feature is an increment of program functionality. Due to the fact that features are implemented through one or more classes, adding a feature subsequently means to introduce codes, namely adding new classes, adding new members and extending members of existing classes[Kuh07]. In the next subsection, we will explain this kind of increment fashion in detail. As the code of classes associated to one feature is merged into one feature module, the terms *feature* and *feature module* synonymously are used for simplicity[AB06].

2.2.2 Feature Oriented Design

Feature oriented design synthesizes individual programs by composing feature modules. One implementation approach of FOP is AHEAD introduced by Batory. In AHEAD, adding features incrementally in a style of stepwise refinement[Bat06]. Each feature corresponds to a layer, which contains a set of partial definitions of classes that implement the feature[BSR03]. A base program is part of every composed product and can be refined stepwise by further feature modules. Then, we can successively compose such layers to generate concrete programs based on user-defined order in configuration file.

For instance, we define the following features for the *Queue* program:

1. *Base*, which implements a base program with enqueue and dequeue functions,
2. *Head*, which returns the first element of a queue without removing it,
3. *Length*, which records the length of queue.

Figure 2.3 depicts the three layers of the *Queue*. Each feature module is stored in one layer. *Base*, *Head* and *Length* are also the names of the features.

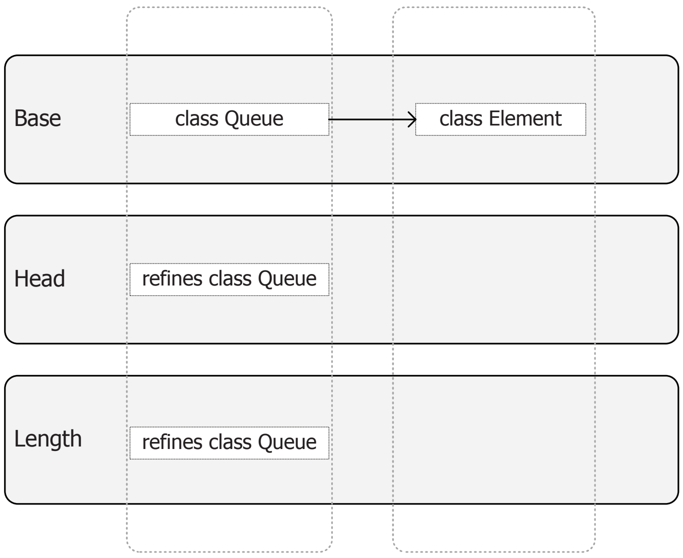


Figure 2.3: Three Features of the *Queue* Program

To illustrate the process of every refinement, we show an example with class *Queue* in Figure 2.4. We assume that the three feature modules, *Base*, *Head* and *Length*, are selected in a configuration process and composed in a top-down order. Class *Queue* is defined with enqueue and dequeue methods in feature module *Base*. Firstly, feature module *Head* refines class *Queue* by adding method *getHead* with keyword "**refines**". Secondly, the Count.jak in feature module *Length* refines class *Queue* by adding field *count* and method *getLength*. Existing methods are extended by overriding, e.g. method *enqueue* and **dequeue** of class *Queue* via an inheritance-like mechanism. These refinements add statements and call the refined method using keyword "**Super**". In general, when feature modules are selected, the features that extend other classes must be defined with the keyword "**refines**" and methods can be refined with keyword "**Super**".

Finally, through composing Jak refinement by the jampack tool of the AHEAD tool suite, we can achieve the result of the composition of feature modules. Figure 2.5 shows the final *Queue* class with the features *Base*, *Head* and *Length*.

```
class Queue{
  private LinkedList elements = new LinkedList();

  public void enqueue(Element element){
    elements.add(element);
  }
  public Element dequeue(){
    elements.removeFirst(element);
  }
}
```

(a) Base.jak

```
refines class Queue{
  public Element getHead(){
    return elements.getFirst();
  }
}
```

(b) Head.jak

```
refines class Queue{
  private LinkedList elements = new LinkedList();
  private int count;

  public Element getHead(){
    if(count>0) Super.getHead();
  }
  public void enqueue(Element element){
    Super.enqueue(element);
    count++;
  }
  public Element dequeue(){
    Super.dequeue();
    count--;
  }
  public int getLength(){
    return count;
  }
}
```

(c) Count.jak

Figure 2.4: Source code examples in Jak of the Queue implementation

```
class Queue{
  private LinkedList elements = new LinkedList();
  private int count;

  public Element getHead(){
    if(count>0)return elements.getFirst();
  }
  public void enqueue(Element element){
    elements.add(element);
    count++;
  }
  public Element dequeue(){
    elements.removeFirst(element);
    count--;
  }
  public int getLength(){
    return count;
  }
}
```

Figure 2.5: The final Queue class refined by features Base, Head and Length

2.3 Refactoring Feature Modules

Refactoring feature modules are a combination of refactorings and feature oriented design. They adjust members and classes by composing feature modules[KBA09]. This section is the basis of the implementation of our thesis. Therefore, we explain the concept of refactoring feature modules and introduce the implementation mechanism of refactoring feature modules in detail.

2.3.1 The Concept of Refactoring Feature Modules

The refactoring feature modules (RFMs) are feature modules that encapsulate refactoring units and integrate refactorings into feature-oriented design[KBA09]. By using this way, we can gain synergy effects. The core concept is to define refactorings in refactoring units that are encapsulated in feature modules[KBA09]. Similar to sequences of feature modules composed to build programs, the defined sequence of refactorings from RFMs can be applied to a program. In other words, the compiler can successively compose the selected features (normal feature or refactoring modules) according to a user-defined order.

Different refactorings expose corresponding parameters whose values define the initial program elements that are to be refactored as well as the target program elements to be refactored into. For example, the parameters of a “Rename Method” refactoring include a qualified name of the method to rename and a new method name. In common IDEs, these parameter values are provided by user selecting code and answering GUI forms. Similarly, in RFMs, a refactoring interface corresponds to a refactoring template. The refactoring interface contains getter methods for acquiring parameters of the corresponding specified refactoring. A refactoring unit is a class-like entity that implements a refactoring interface. The getter methods of the refactoring interface are implemented by the refactoring unit then each method returns a value for that parameter. By using this way, a concrete refactoring can be specified by setting the parameter values of a refactoring unit.

To illustrate, we depict two samples, which encapsulate two refactoring units ”RenameMethodEnqueueToAddTail” and ”RenameMethodDequeueToRemoveHead” in Figure 2.6.

Figure 2.7 depicts a sequence of RFMs that is applied successively in a top-down order to the result program of the composition of the three feature modules in the preceding subsection.

Furthermore, the existing work about the RFMs not only focused on transformations that monotonically add code in order to produce program transformations [KBA09], but also can guarantee that refactorings and sequence of refactorings are composed without errors in feature oriented designs[KBK09b].

2.3.2 The Tool Support of Refactoring Feature Modules

Due to the requirement of implementing presented concepts in a prototype, the understanding of an existing tool is necessary. The implementation of concepts of the existing

```
refactoring RenameMethodEnqueueToAddTail implements RenameMethodRefactoring {
  public String getOldMethodName(){
    return "Queue.enqueue";
  }
  public String getNewMethodName(){
    return "addTail";
  }
}
```

(a) RenameMethod Refactoring From ‘Enqueue’ To ‘AddTail’

```
refactoring RenameMethodDequeueToRemoveHead implements RenameMethodRefactoring {
  public String getOldMethodName(){
    return "Queue.Dequeue";
  }
  public String getNewMethodName(){
    return "removeHead";
  }
}
```

(b) RenameMethod Refactoring From ‘Dequeue’ To ‘RemoveHead’

Figure 2.6: Rename Method Refactoring

```
class Queue{
  private LinkedList elements = new LinkedList();
  private int count;

  public Element getHead(){
    if(count>0)return elements.getFirst();
  }
  public void addTail(Element element){
    elements.add(element);
    count++;
  }
  public Element dequeue(){
    elements.removeFirst(element);
    count--;
  }
}
```

(a) enqueue→addTail

```
class Queue{
  private LinkedList elements = new LinkedList();
  private int count;

  public Element getHead(){
    if(count>0)return elements.getFirst();
  }
  public void addTail(Element element){
    elements.add(element);
    count++;
  }
  public Element removeHead(){
    elements.removeFirst(element);
    count--;
  }
}
```

(b) dequeue→removeHead

Figure 2.7: The sequence of RFMs

tool can be seen as an extension to the Jak language which extends support for feature modules [BSR03]. The technical report [KBA08]describes a plugin mechanism to support different kinds of refactorings for refactoring units where one refactoring corresponds to one plugin. Refactoring plugins follow a common interface and are loaded by the compiler when corresponding refactoring units are executed. Refactoring interfaces are declared inside every plugin and each interface corresponds to one type of refactorings. Then a refactoring interface is implemented by corresponding refactoring units during the compilation.

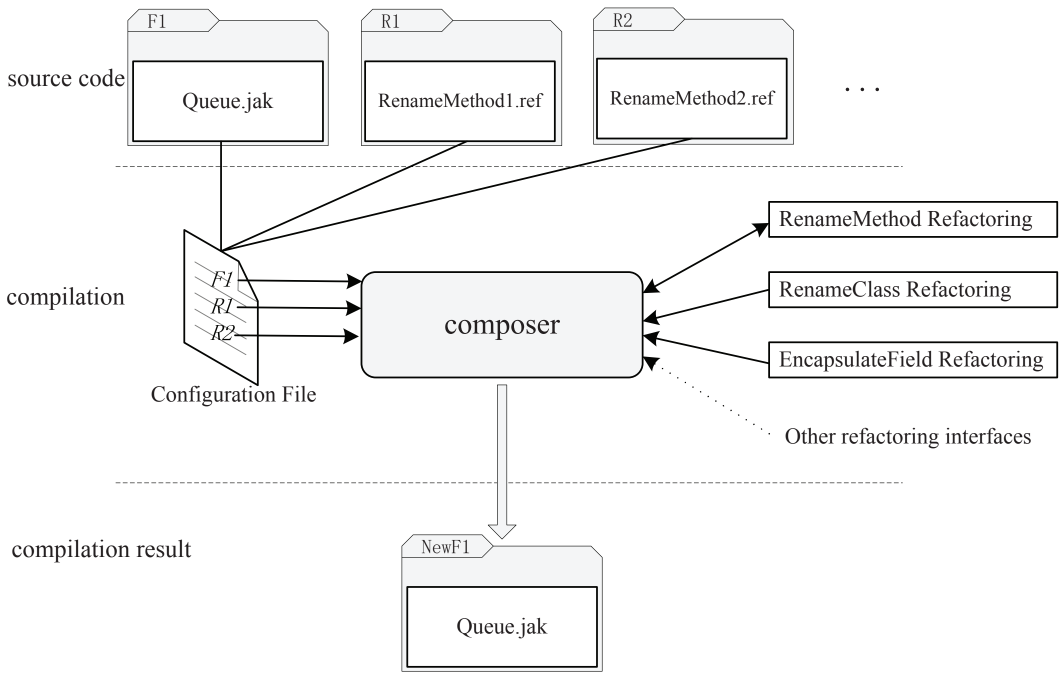


Figure 2.8: Implementation of RFMs

As illustrated in Figure 2.8, before the compilation, we assume there is a simple FOP program: *Queue.jak*, the content of which was shown in layer **F1** in Figure 2.5. At the same time, we define two "Rename Method" refactorings by implementing the corresponding interfaces in two refactoring units: *RenameMethodEnqueueToAddTail.ref* and *RenameMethodDequeueToRemoveHead.ref*, the content of which were shown in Figure 2.6(a) and Figure 2.6(b) respectively. In Figure 2.8, we use *RenameMethod1.ref* and *RenameMethod2.ref* instead of *RenameMethodEnqueueToAddTail.ref* and *RenameMethodDequeueToRemoveHead.ref* for short respectively. Moreover we assume that the user-defined order of the sequence of RFMs is written in the configuration file when selected features are **F1**, **R1** and **R2**.

During the compilation, the existing tool (composer) can successively compose the selected feature modules and refactoring feature modules based on the user-defined order in configuration file.

After the compilation, the transformed program *Queue.jak* is generated successfully.

In the following chapters, we will come up with one methodology for optimizing the sequence of refactorings loaded during compilation. If we find out the optimized sequence of refactorings, we shall rewrite the refactoring units and the execution order of refactorings to the configuration file before compilation.

2.4 Summary

This chapter began with the concept of refactoring and the composition of refactorings. These concepts inspire us to analyze potential relationships among refactorings. In addition, the application of refactoring feature modules in feature oriented programming is the basis of the thesis. There were descriptions of two important concepts, for one thing, we reviewed the concept of feature and the principle of feature oriented design, For another, we briefly introduced the concept of refactoring feature modules and the application of refactoring feature modules based on existing tool.

Chapter 3

Principle of Optimizing Sequences of Refactorings

In practical applications, after users select refactoring feature modules (RFMs) and define the execution order of them, these RFMs are sequentially composed to a base program by compilation of existing compose tool. During these RFMs are composed, corresponding refactorings encapsulated in these RFMs are applied to the base program. Therefore, the key to reduce the compilation time of RFMs is to optimize the sequences of refactorings.

In this chapter, we provide a complete description of our approach about optimizing sequences of refactorings. Firstly, we start with a detailed description of refactorings and corresponding operations in Section 3.1 and then we explore the potential relationships among refactorings within user-defined sequences of refactorings in Section 3.2. In addition, we define the relevant basic notations on which we depend in order to theoretically express our approach in Section 3.3. Last but not least, in Section 3.4 we describe concepts of logical optimization and physical optimization respectively.

3.1 Refactorings

In general, a refactoring corresponds to a behavior-preserving conditional transformation performed on a given program[KK04]. From our perspective, the given program consists of different types of program elements such as class, method, field and other group of cohesive codes. When we apply a series of refactorings encapsulated in RFMs to the given program, the program structure can be sequentially transformed by operating relevant program elements but the program functionality are unchanged.

Essentially, a typical refactoring application can be divided into two phases:

In the preconditions checking phase, if program elements of the given program satisfy conditions of a refactoring, we can enable operations of this refactoring to operate relevant program elements. Before a refactoring is executed, the process of checking whether its conditions are satisfied is called preconditions checking. The preconditions checking of a refactoring requires searching the whole program to determine whether it is legal. It is a global concept and need to consider the whole program.

In the transformation phase, when its preconditions are satisfied, the operation of refactorings to relevant program elements can be executed. The refactoring operation transforms relevant program elements to the target program elements. It is a local concept and reflects transformation effects of refactorings.

As illustrated in Figure 3.1, the "rename method" refactoring renames the method name "Queue.foo" to the new method name "Queue.enqueue". In the preconditions checking phase, the preconditions checking of this specified refactoring requires to search the whole program and check the existence of the method "Queue.foo" and the non-existence of the "Queue.enqueue". In the transformation phase, if its preconditions are satisfied, the transformation effect is reflected by operating relevant identifiers. In the thesis, identifiers are used to indicate the signatures of program elements. In practical application, the identifiers of specified operation of a refactoring are similar to the parameter values of the specified refactoring. For example, the "rename method" refactoring renames the old method name into the new method name and updates all references. There are two corresponding identifiers. One is to indicate the old method name and all references, the other is to indicate the new method name and all updated references.

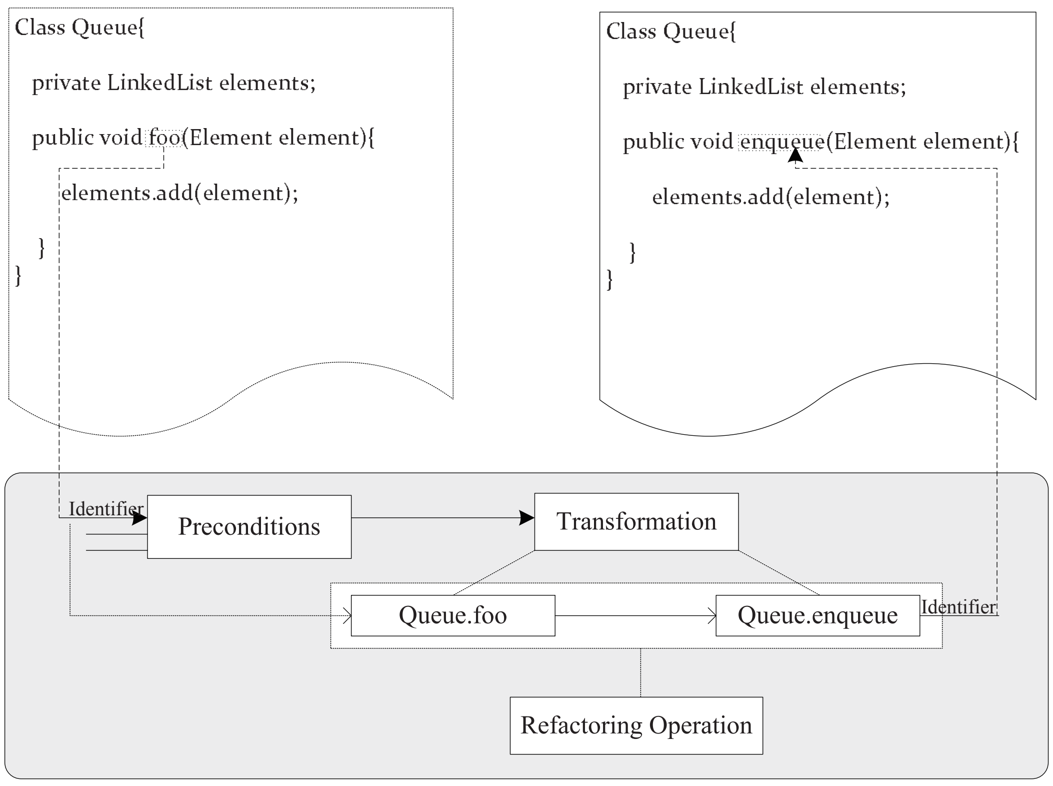


Figure 3.1: An example of "rename method" refactoring

Existing automatic refactoring tools support different types of refactorings by setting different parameters. The specified parameter values correspond to program elements that to be refactored and program elements that are refactored into. For example, the "rename method" refactoring transforms the name of the method in the given program by operating relevant program elements of the method. The parameters of the "rename method" refactoring are the fully qualified name of the initial method and the updated method name. Figure 3.2 depicts a unified and simplified refactoring. As illustrated in

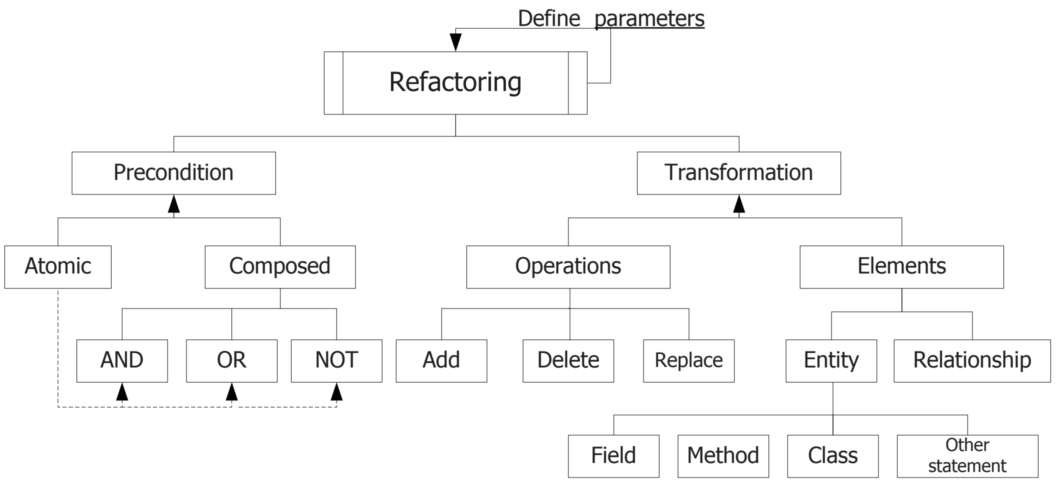


Figure 3.2: Unified model of a typical refactoring

Figure 3.2, in the preconditions checking phase, the precondition may contain atomic conditions or complex ones created by conjunction, disjunction and negation of subconditions. Atomic conditions refer to checking the typical structure of a concrete program. In the transformation phase, Transformations can alter structure of a concrete program by primitive operations (add, delete, replace, etc). For example, the transformation of “rename class” refactoring can be applied by deleting the old class and adding the new class.

Since the number of possible refactorings is unlimited, there is no program tool can integrate all refactorings for various user requirements[KK04]. Fortunately Fowler's list of standard refactorings contains the common refactorings that can satisfy the requirements of most users. From a formal perspective, one refactoring operation can be seen as a unary algebra operator applied to a single expression of refactored program. Based on the sixty eight different types of standard refactorings introduced by Fowler and two common ones namely “rename class” refactoring, “rename Field” refactoring[1], we can derive the seventy corresponding basic refactoring operators of the algebra. Table 3.1 depicts the nine typical refactoring operators[2] of them. The names of these refactoring operators correspond to the abbreviations of the standard refactorings.

Theoretically, these refactorings can be implemented in refactoring units encapsulated in the RFMs. Operations of these refactorings modify different types of program elements in order to produce expected program transformation. In fact, these refactorings seldom are used alone, while sequences of refactorings can acquire a more meaningful transformation. In the next section, we explore the characteristics of sequences of refactorings.

[1]These 70 refactorings are called standard refactorings in thesis. 26% of the standard refactorings are covered in existing composition tool of RFMs[KBK09b].

[2]Other operators can be referred in Appendix A.

RefOps [3]	Transformation Description
RF	The “Rename Field” refactoring transforms a name of the field to a new one.
RC	The “Rename Class” refactoring transforms a name of the class to a new one.
RM	The “Rename Method” refactoring transforms a name of the method to a new one.
EI	The “Extract Interface” refactoring extracts the same subset from client class into an interface
IM	The “Inline Method” refactoring replaces a call of old method name with the fragment of code in its body
EF	The “Encapsulate Field” refactoring deals with the public field and make it private and provide accessors for other classes.
MM	The “Move Method” refactoring moves one method from the source class to the target class.
SA	The “Substitute Algorithm” refactoring replaces the body of the method with the new algorithm.
CH	The “Collapse Hierarchy” refactoring merges superclass and subclass together

Table 3.1: Basic Refactoring Operators

3.2 Sequences of Refactorings

Sequences of refactorings which we optimize are not randomly generated from the predefined refactorings encapsulated in RFMs. They are user-defined sequences of refactorings encapsulated in the RFMs and can be executed in user-selection orders. In order to define relational algebra for computation of optimization for sequences of refactorings, we continue to explore the properties of user-defined sequences of refactorings and reveal potential relationships between refactorings in sequences.

3.2.1 Properties of Sequences of Refactorings

The assumption of our approach is that the sequences of refactorings which we optimize can be executed successfully. In fact, the existing technique has already guaranteed the sequences of RFMs can be composed safely. In other words, the user-defined sequences of refactorings can be guaranteed to be performed successfully by verifying the preconditions of the involved refactorings[KBK09a]. Therefore, the properties of user-defined sequences of refactorings which are given to us for optimization include three aspects:

- All the refactorings are defined in corresponding RFMs
- The execution order of refactorings is defined in the configuration file by user
- The sequence of refactorings must be performed successfully.

[3]RefOps denotes basic refactoring operators.

Technically, for refactorings encapsulated in RFMs, their preconditions can be formulated in terms of identifiers which must exist when an RFM is applied and identifiers which must not exist[KBK09a]. An isolated refactoring's precondition is established in the initial program. But if a refactoring is in a legal sequence of refactorings, its preconditions might be established in the initial program or set up by preceding refactorings. This is an important property of legal sequences of refactorings. If the preconditions of refactorings in a given sequence of refactoring can be satisfied by execution of the preceding refactorings, there could be chances for optimization.

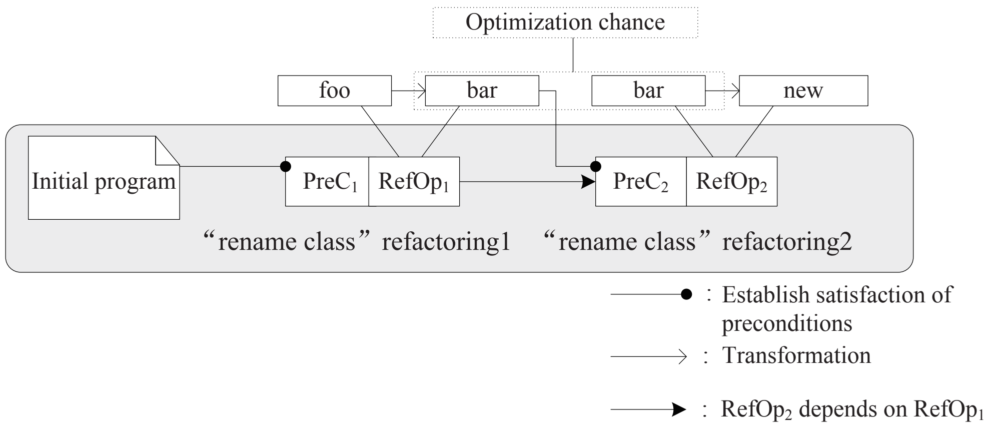

Optimization chance
foo
bar
bar
new
Initial program
PreC1
RefOp1
PreC2
RefOp2
"rename class" refactoring1
"rename class" refactoring2
: Establish satisfaction of preconditions
: Transformation
: RefOp2 depends on RefOp1

Figure 3.3: Two kinds of establishing satisfaction of preconditions

As illustrated in Figure 3.3, the $RefOp$ denotes a general refactoring operation and the $PreC$ denotes precondition of corresponding refactoring. There are two refactorings in the user-defined sequence. The "rename class" refactoring which renames a class *bar* into *new* requires program elements with identifier "bar" to exist and program elements with identifier "new" not to exist.

The first precondition of this refactoring can be set up by preceding refactorings. For example, the preceding 'rename class" refactoring renames a class *foo* into *bar*. This preceding refactoring establishes the identifier 'bar' of the later refactoring, in other words, the later refactoring depends on the preceding refactoring. In our concept, this dependency relationship is called set-up-identifier dependency relationship. This kind of dependency relationship provides an chance for optimization. We can merge the two refactorings to one refactoring which renames "*foo*" to "*new*".

As a part of our concept, the second precondition of this refactoring also can be set up by preceding refactorings. For example, if there is the preceding 'rename class' refactoring which renames a class 'new' into 'other'. This preceding refactoring establishes one deletion of identifier 'new' of the later refactoring. In this situation, the later refactoring depends on the preceding refactoring. But this kind of dependency relationship do not expose optimization potential, while we should keep the order of the two refactorings in sequence during optimization process.

In summary, for legal user-defined sequences of refactorings, if the preconditions of a refactoring are influenced by the preceding refactorings, there is dependency relationship among refactorings in the sequence. Based on the dependency relationships, we can expose optimization potentials for user-defined sequences. In the next section, the types of relationship between refactorings in legal user-defined sequences will be explored.

3.2.2 Potential Relationships of Refactorings

The user-defined sequence of refactorings can be thought of as a legal-guaranteed sequence that is made up of a set of refactorings. In this section, we describe two categories of potential relationships of refactorings by analyzing the properties of sequences of refactorings as follows:

1. The commutative relationship. If the preconditions of the individual refactorings in a sequence are not related in any way, we can swap the location of any two refactorings in the sequence. In such situation that if the locations of two refactorings are interchangeable, we call that there is the commutative relationship between the two refactorings.

   Based on this kind of relationship, it is no doubt that we can rearrange the locations of relevant refactorings.

2. The dependency relationship. If the preconditions of the refactorings influence the satisfaction of preconditions of the following refactorings in a sequence, we cannot directly swap the locations of any two refactorings and usually require keeping the order of involved refactorings in the sequence. In such situation that if the preconditions of the refactoring depend on the transformation of the previous refactoring in a pair of refactorings, we call that there is a dependency relationship between the two refactorings.

   In our concept, the dependency relationship can be further divided into three types of dependency relationship as follows:

   First is set-up-identifier dependency where one refactoring establishes required identifiers for a later refactoring. Based on set-up-identifier dependency, we can consider detecting whether there are optimization potentials among involving refactorings. If the involving refactorings can be optimized, they could be merged to an updated refactorings by applying optimization rules. If the involving refactorings cannot be optimized, their order should be kept and disallow being reordered.

   Second is set-up-part-of-identifier dependency where one refactoring establishes part of an identifier for a later refactoring. Due to the fact that identifier is fully qualified name of program elements, the fully qualified name of a method consists of identifier of class and the method name. In other words, the part of identifier of a method could be a hosting class name. Therefore, one refactoring could establish part of an identifier for a later refactoring. For example, there are two refactorings, 'rename class' refactoring which renames 'foo' to 'bar' and 'rename method' refactoring which renames 'bar.old' to 'bar.new'. The former refactoring establishes part of identifier to the latter refactoring. Based on set-up-part-of-identifier dependency, we consider commute the involving refactorings to expose optimization potentials. But the execution of commutation operation requires that the relevant refactorings update their parameters for guaranteeing that the reordering sequence is a legal sequence.

   Third is set-up-deletion-of-identifier dependency where one refactoring establishes required deletions for a later refactoring. Based on set-up-deletion-of-identifier dependency, we could not expose the optimization potential. Even if we found that the involved refactorings have optimization potential with others, we still need to keep the order of the involved refactorings and disallow reordering them.

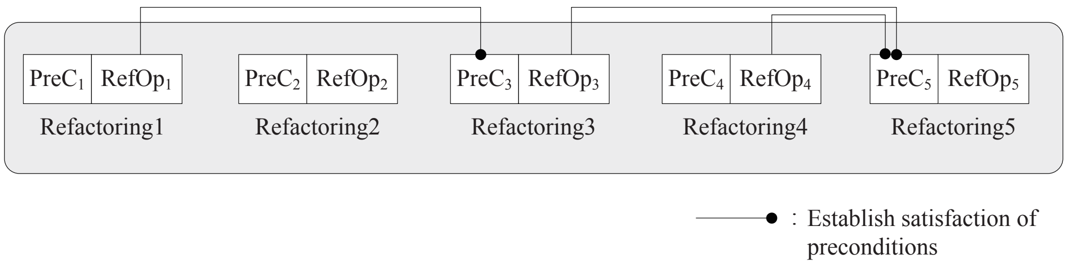


Figure 3.4: Potential Dependency Relationship

To illustrate, Figure 3.4 depicts the potential dependency relationships in a sample sequence of refactorings. The three dependency relationships between $Refactoring1$ and $Refactoring3$, $Refactoring3$ and $Refactoring5$, $Refactoring4$ and $Refactoring5$ are shown. Unless specifically indicated, there is commutative relationship between each other. In practice, both the commutative relationship and the dependency relationship are potential relationships in a user-defined sequence. The two kinds of potential relationships, especially the dependency relationships, support our approach. We should firstly expose the two kinds of relationships and separate the initial sequence into different chains of refactorings[4].

3.3 Formal Perspective

Before explaining our approach, we want to clarify some terms and notations for formal expression of sequences of refactoring operations and to extend relevant concepts about analysis of optimization in the remainder of this section.

3.3.1 Definition of Refactoring Operations

In general, refactorings are defined as program transformations that have preconditions that must be satisfied before the transformation is performed successfully. Roberts' dissertation[Rob99] extends this concept to support computing dependencies between refactorings. In his work, he specified preconditions with first order predicate calculus. By using this way, we can formalize the dependencies between refactorings. As we mentioned in the preceding sections, the preconditions of every refactorings in sequences of refactorings are satisfied before we optimize the sequences. Due to preconditions of a refactoring have been satisfied, we focus on the operation of the refactoring. For our approach, we firstly extend the concept of refactorings and then give a special definition.

We use $RefOp$ to denote one general refactoring operation and the abbreviation of the name of standard refactorings to denote the specific refactoring operations. Besides we introduce a pair of terms consists of $PreKeys$ and $PostKeys$. The concept

[4]A chain of refactorings corresponds to a sequence in which there are dependency relationships between any two refactorings.

of *RefOp*'s *PreKeys* is derived from identifiers that need to exist for satisfaction of its preconditions before this refactoring is executed. When other preconditions are also satisfied, these identifiers will be operated by this refactoring. The concept of *RefOp*'s *PostKeys* is derived from identifiers that are transformed after we execute the refactoring. In essential, for refactorings encapsulated in RFMs, the preconditions of some refactorings require to analyze the properties of inheritance hierarchies[KBK09a]. We do not consider these kinds of preconditions by checking concrete programs, because these kinds of preconditions should be satisfied in the process of the implementation of existing tool. Besides, if their preconditions are formulated in terms of identifiers which must exist when an RFM is applied and identifiers which must not exist, we define the identifiers, which must exist and to be transformed, to be *PreKeys*, and define the transformed identifiers to be *PostKeys*.

For example, the “rename class” refactoring renames a class “List” into “Queue”. The preconditions include both that program elements with identifier “List” to exist and that program elements with identifier “Queue” not to exist. As for our approach, we only derive the identifier “List” that must to exist from the preconditions as the *PreKey*. The *PreKey* of this refactoring operation is “List” and its *PostKey* is “Queue”, which is derived from the transformed program elements with one identifier “Queue”. Moreover, if the type of program elements is method or field, the identifier of the method or the field is the fully qualified name which can indicate the corresponding program elements. The fully qualified name consists of the name of the class and the name of the method or the name of the field. For instance, the “rename method” refactoring which renames a method “foo” into “enqueue”. One of its preconditions is that identifier “Queue.foo” to exist. After performing this refactoring, the transformed program elements with identifier is “Queue.enqueue”. Thus, the *PreKeys* of “rename method” refactoring is “Queue.foo” and the *PostKeys* is “Queue.enqueue”. Due to all preconditions and transformations of refactorings are satisfied before we optimize them, we only focus on the relevant program elements with identifiers.

Besides, some other types of refactorings have more than one identifier as *PostKeys* or *PreKeys*. For example, the “Encapsulate Field” refactoring which encapsulates field “index” in class “Queue”. These relevant program elements about this field are indicated by identifier “Queue.index”. Its preconditions contain the identifier “Queue.index” that must exist, the access specifier of field “index” that is not private and more other conditions. We focus on one of the preconditions is field ”Queue.index” must exist. Thus the *PreKey* of this “Encapsulate Field” refactoring is the identifier ”Queue.index”. On the other hand, the transformed program elements of this refactoring usually contain the access specifier of field “index”, the setting method “setIndex” for modifying the value of field “index” and the getting method “getIndex” for accessing the value of field “index”. Thus, the *PostKeys* consists of the identifier “Queue.index”, identifier “Queue.getIndex” and identifier “Queue.setIndex”.

Therefore, when preconditions of a refactoring are satisfied, we define the refactoring operation with *PreKeys* and *PostKeys* for our approach.

Definition 3.1 *In user-defined sequences of refactorings, a refactoring operation is* $RefOp_{PreKeys \Rightarrow PostKeys}$*, where PreKeys are derived from program elements that to be transformed and PostKeys are derived from identifiers of updated program elements.*

Based on this definition, the "rename class" refactoring which renames a class "List" into "Queue" can be described as $RC_{List \Rightarrow Queue}$, the "rename method" refactoring which renames a method "foo" into "enqueue" can be described as $RM_{Queue.foo \Rightarrow Queue.enqueue}$ and "Encapsulate Field" refactoring which encapsulates field "index" in class "Queue" can be described as $EF_{Queue.index \Rightarrow Queue.index|Queue.getIndex|Queue.setIndex}$. In practical application, our optimization probably consider one or more identifiers from the $PreKeys$ and the $PostKeys$. So we use "|" to denote the disjunction of identifiers in the $PreKeys$ and the $PostKeys$.

RefOps	Transformation Description
$RF_{C.f \Rightarrow C.newf}$	The "Rename Field" refactoring transforms a name of the field($C.f$) to a new one($C.newf$).
$RC_{C \Rightarrow newC}$	The "Rename Class" refactoring transforms a name of the class(C) to a new one($newC$).
$EI_{C \Rightarrow I}$	The "Extract Interface" refactoring extracts the same subset from client class(C) into an interface(I)
$IM_{M \Rightarrow \ldots}$	The "Inline Method" refactoring replaces a call of old method(M) with the fragment of code in its body(...)
$RM_{C.M \Rightarrow C.newM}$	The "Rename Method" refactoring transforms a name of the method($C.M$) to a new one($C.newM$).
$EF_{C.f \Rightarrow C.f|C.getf|C.setf}$	The "Encapsulate Field" refactoring deals with the public field($C.f$) and make it private and provide accessors for other classes.
$MM_{C1.M|C2 \Rightarrow C1|C2.M}$	The "Move Method" refactoring moves one method(M) from the source class($C1$) to the target class($C2$).
$SA_{C.M \Rightarrow C.M}$	The "Substitute Algorithm" refactoring replaces the body of the method($C.M$) with the new algorithm.
$CH_{C1|C2 \Rightarrow C1}$	The "Collapse Hierarchy" refactoring merges superclass($C1$) and subclass($C2$) together($C1$)

Table 3.2: Formal expression for basic refactoring operators

As illustrated in Table 3.2, we derive the formal expressions for nine refactoring operations to extend the Table 3.1. In the following descriptions, we will use these formal expressions to explain the refactoring operations for short.

3.3.2 Definition of Sequences of Refactoring Operations

The refactorings encapsulated in RFMs are rarely performed in isolation but performed in sequence to acquire a more interesting transformation. In RFMs, the user-defined sequences of refactoring operations can be defined as follows:

Definition 3.2 *A user-defined sequence of refactoring operations is <$RefOp_1$, $RefOp_2$, $RefOp_3$, ..., $RefOp_n$>, where these $RefOps$ are sequentially applied to a program successfully. $RefOp_i$ is the refactoring operation in the ith location of the sequence, the order of execution is left to right and the length of the sequence is n.*

To illustrate, there is a legal three-refactoring-operation sequence $<RC_1, RM, RC_2>$, where $RC_{1_{C1 \Rightarrow C2}}$, $RM_{C2.M1 \Rightarrow C2.M2}$, $RC_{2_{C2 \Rightarrow C3}}$. The first refactoring operation is to rename the class $C1$ to class $C2$, the second is to rename the method $C2.M1$ to $C2.M2$, and the third is to rename the class $C2$ to $C3$.

3.3.3 Definition of Relational Operations

From a formal perspective, a binary algebra operator is applied to two expressions. Based on the relationships among refactoring operations in sequences of refactoring operations, we define four binary relational operators to operate two expressions of refactoring operations in this section.

If there are potential commutative relationships and dependency relationships in a given sequence of refactoring operations, we need to expose commutative relationships and dependency relationships by separating different chains of refactorings. In this section, we firstly define two commutative operations to separate the chains of refactoring operations as follows:

Definition 3.3 *commutative operator "↔"*
If there is a commutative relationship between the two refactoring operations in a given sequence: $<RefOp_1, RefOp_2>$, we can swap locations of the $RefOp_1$ and $RefOp_2$. This kind of commutative laws is shown as:
$RefOp_1 \leftrightarrow RefOp_2 = RefOp_2 \leftrightarrow RefOp_1$

In our concept, the application of the commutative operator "↔" is based on the commutative relationships among the refactorings. To accurate, there is a commutative relationship between the two refactoring operations, namely the $RefOp_1$ and $RefOp_2$. They modify mutually non-related program elements and their locations can be reordered. Changing the execution order of the two refactorings does not influence the results of final transformation. Take the pair of refactoring operations $<RM, RF>$ where $RM_{C1.M1 \Rightarrow C1.M2}$ and $RF_{C1.F1 \Rightarrow C1.F2}$ as an example. If we swap the locations of RM and RF, we can get the pair of refactoring operations $<RF, RM>$. At the same time, the transformation results of the pair of refactoring operations $<RM, RF>$ is equivalent to the original pair.

The commutative operator is unique relational operator that is based on the commutative relationship among refactorings inside sequences of refactorings. Furthermore, we can extend the concept of commutative operator that is based on the dependency relationship. To accurate, this kind of commutative operator is based on set-up-part-of-identifier dependency.

To illustrate, if one of the identifiers of program elements modified by the $RefOp_1$ contains one of the identifiers of program elements modified by the $RefOp_2$, we should update the identifier of program elements of $RefOp_1$ before swapping the locations of the $RefOp_1$ and the $RefOp_2$. For example, There is a pair of refactorings $<RM, RC_2>$ in three-refactoring-operation sequence $<RC_1, RM, RC_2>$, where $RC_{1_{C1 \Rightarrow C2}}$, $RM_{C2.M1 \Rightarrow C2.M2}$, $RC_{2_{C2 \Rightarrow C3}}$. There is an optimization chance to optimize $<RC_1, RC_2>$. Hence, we want to swap the locations of RM and RC_2, we should firstly update the relevant identifiers in the $PreKeys$ and $PostKeys$ for the RM. Then we can get the legal sequence $<RC_1, RC_2, RM>$, where $RC1_{C1\Rightarrow C2}$, $RC2_{C2\Rightarrow C3}$, $RM_{C3.M1\Rightarrow C3.M2}$. In order to solve this situation about set-up-part-of-identifier dependency, we introduce the definition of the conditional commutative operator as follows:

Definition 3.4 *conditional commutative operator "$\Leftrightarrow$"*
If there is a set-up-part-of-identifier relationship between the two refactoring operations in a given sequence: $<RefOp1_{PreKey1\Rightarrow PostKey1}, RefOp2_{PreKey2\Rightarrow PostKey2}>$, we can swap locations of the $RefOp_1$ and $RefOp_2$. This kind of commutative laws is shown as:
$RefOp1_{PreKey1\Rightarrow PostKey1} \Leftrightarrow RefOp2_{PreKey2\Rightarrow PostKey2} =$
$RefOp2_{PreKey2'\Rightarrow PostKey2'} \Leftrightarrow RefOp1_{PreKey1\Rightarrow PostKey1}$

Based on application of "$\leftrightarrow$" operator and "$\Leftrightarrow$" operator, we can separate the sequence of refactoring operations to different chains of refactoring operations by slightly reordering locations of refactoring operations. By using this way, we can get a new sequence consisting of different chains of refactoring operations. In general, the new equivalent sequences can expose potential chances to be optimized.

For a general chain of refactorings which may be of any length, we can simplify the computation of its full precondition and transformation by analyzing the precondition of each refactoring in this chain and corresponding transformation effects. By using this way in the extreme situation, we can achieve a composite refactoring for the entire chain. In our approach, we start to analyze the preconditions and transformation for pair of refactorings in the chain step by step. There exists complexity in automatically eliminating the preconditions and merging the transformations. We manually explore the optimization rules for pair of refactorings. The basic concepts of the optimization effect are based on eliminating the preconditions and merging or replacing the transformations.

Based on these concepts, we define two operators to express the ways of processing the pairs of refactoring operations within a chain of refactoring operations for optimization effect as follows:

Definition 3.5 *Each chain of refactorings is a sequence of refactoring operations $< RefOp_1\ \theta\ RefOp_2\ \theta\ RefOp_3\ \theta\ \ldots\ \theta\ RefOp_n>$ where each RefOp is executed successfully. "θ" denotes the operator in $\{\rightarrow, \Rightarrow\}$ which can be applied for pairs of refactoring operations in the optimization analysis.*

Given a pair of refactoring operations $<RefOp_1, RefOp_2>$ from a chain of refactoring operations, $RefOp_1 \rightarrow RefOp_2$, $RefOp_1$ sets up preconditions of $RefOp_2$ and $RefOp_2$ depends on $RefOp_1$. In other words, the one or more $PostKeys$ of $RefOp_1$ are identical to the only one or multiple $PreKeys$ of $RefOp_2$.

In this situation, there is optimization chance for the $RefOp_1$ and $RefOp_2$ in the theoretical level. In practical application, if we can implement the optimization effect for both $RefOp_1$ and $RefOp_2$, we use relational operator "$\Rightarrow$" to denote that the $RefOp_1$ and $RefOp_2$ can be optimized. The optimization effect includes two possible ways:

- The "$RefOp_1 \circ RefOp_2$" denotes the composition of transformations. Take $<RM, MM>$ for instance, where RM renames the method $C1.M1$ into the $C1.M2$, MM moves the method $M2$ from class $C1$ into class $C2$. The transformation of RM generates the method $C1.M2$, which guarantees that the method $C1.M2$ have already existed. As an independent refactoring, one of the preconditions of MM need to check that the method $C1.M2$. After composing $<RM, MM>$, the cost of checking overall preconditions $<RM \circ MM>$ is lower than checking the preconditions for these refactorings respectively[Rob99]. So we can compose these two refactoring operations to a composite refactoring operation $<RM \circ MM>$ by merging the preconditions and composing the transformations. This kind of optimization way has already been proposed by existing researches[Rob99][KK04] .

- The "$RefOp_3$" denotes the simplification of transformations. Take $<RM_1, RM_2>$ for instance, where RM_1 renames the method $C1.M1$ into the $C1.M2$, RM_2 renames the method $C1.M2$ into the $C1.M3$. The preconditions and transformation of RM_1 guarantee the all preconditions of RM_2. We can replace the pair of refactoring operations $<RM_1, RM_2>$ by $<RM_3>$ which renames the method $C1.M1$ into the $C1.M3$. The preconditions of RM_1 can guarantee the execution of RM_3. This kind of optimization way is an important point of our concept.

Although there is a dependency relationship between $RefOp_1$ and $RefOp_2$, if we cannot implement the optimization effect for both $RefOp_1$ and $RefOp_2$. we must not break the relationship of $RefOp_1$ and $RefOp_2$ and should keep the order of the $RefOp_1$ and the $RefOp_2$. We can use the relational operator "$\rightarrow$" to denote the keeping order of the $RefOp_1$ and the $RefOp_2$. For instance, there is a three-refactoring-operation sequence $<MM, RM, SA>$ where MM moves the method $M1$ from class $C1$ into class $C2$, RM renames the method $C2.M1$ into the $C2.M2$ and SA changes the algorithm for the method $C2.M2$. In our concept, we predefine the optimization rules for pair of refactoring operations. So far, it is impossible to compose the three refactoring operations together $<MM \circ RM \circ SA>$. The alternative solution could be $<MM \circ RM, SA>$ or $<MM, RM \circ SA>$. We should keep the initial order of the unoptimized refactoring operation and the composite refactoring operation.

Another instance is a four-refactoring-operation sequence $<RC_1, RC_2, RC_3, RC_4>$ where $RC1_{A \Rightarrow B}$, $RC2_{Z \Rightarrow Y}$, $RC3_{B \Rightarrow C}$ and $RC4_{C \Rightarrow Z}$. There are two original classes A and Z. The locations of the RC_2 and the RC_4 also should keep and disallow to be reordered. Because there exists a *set-up-deletion-of-identifier dependency. set-up-deletion-of-identifier dependency* is a special dependency relationship where a refactoring establishes a required deletion of a later refactoring. In this instance, if the class Z is not renamed to class Y, the $R4$ cannot be executed because it is impossible that there are two identical classes in Java and Java-like languages. By using this concept, we can avoid reordering some refactorings whose the name of $PreKey$ and $PostKey$ is identical to the name of updated program elements. If we reorder the location of RC_4 with RC_2, satisfaction of its preconditions will be broken. Hence, we should disallow reordering the location of RC_4.

3.3.4 Optimization Rules

Based on refactoring operators and relational operators, we can heuristically reveal several optimization rules for any pairs of refactoring operations within a chain of refactoring operations. These optimization rules play a central role in identifying alternative sequences of refactoring operations. Given a pair of refactoring operations, if one or more $PostKeys$ of the first refactoring operation is identical to one or more $PreKeys$ of the second refactoring operation, it is possible that we reveal optimization rules for this pair of refactoring operations. In practical application, the optimization rules can be classified into two types of rules: simplification rules and composition rules.

1. Simplification rules:
In general, for a pair of refactoring operations: $<RefOp_1$, $RefOp_2>$ in the chain of refactoring operations, if the $PostKeys$ of the first refactoring are identical to the $PreKeys$ of the second refactoring and there is the transformation of one alternative refactoring instead of the transformation of the pair of refactorings, we can apply the simplification rules to the pair of refactorings.

The graphical depiction for unified simplification rules is illustrated in Figure 3.5. The former refactoring's transformation and preconditions can provide all the satisfaction of the latter refactoring's preconditions. So the former refactoring's preconditions are enough for performing the alternative refactoring whose transformation is derived from the original two transformations by merging operations.

Simplification Rules:

$[\![\langle \text{PreC1, Transformation1} \rangle \Rightarrow <\text{PreC2, Transformation2}> \equiv \langle \text{PreC1, Transformation13} \rangle]\!]$

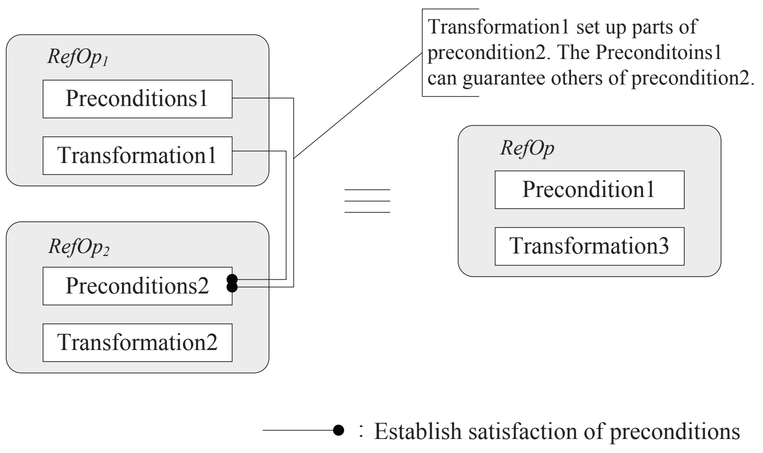

Transformation1 set up parts of precondition2. The Preconditoins1 can guarantee others of precondition2.
RefOp1
Preconditions1
Transformation1
RefOp2
Preconditions2
Transformation2
RefOp
Precondition1
Transformation3
: Establish satisfaction of preconditions

Figure 3.5: Unified simplification rules

Example 1: There is a pair of refactorings $< RF_1, RF_2 >$, where $RF1_{C.f1 \Rightarrow C.f2}$, $RF2_{C.f2 \Rightarrow C.f3}$. It means two rename field refactorings following each other, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of RF_1 sets up one of the preconditions of RF_2: the field $C.f2$ must exist. The preconditions of the former refactoring are enough to guarantee both transformations. We can replace both transformations by an alternative transformation. The new refactoring consists of the precondition of the former refactoring and the alternative transformation. $<RF_1 \Rightarrow RF_2>\equiv<RF_{new}>$, where $RF_{new_{C.f1 \Rightarrow C.f3}}$.

Example 2: There is a pair of refactorings $<RC_1, RC_2>$, where $RC1_{C1 \Rightarrow C2}$, $RC2_{C2 \Rightarrow C3}$. It means two rename class refactorings following each other, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of RC_1 sets up one of the preconditions of RC_2: the class $C2$ must exist. The preconditions of the former refactoring are enough to guarantee both transformations. We can replace both transformations by an alternative transformation. The new refactoring consists of the precondition of the former refactoring and the alternative transformation. $<RC_1 \Rightarrow RC_2> \equiv <RC_{new}>$, where $RC_{new_{C1 \Rightarrow C3}}$.

Example 3: There is a pair of refactorings $< RM_1, RM_2 >$, where $RM1_{C.M1 \Rightarrow C.M2}$, $RM2_{C.M2 \Rightarrow C.M3}$. It means two rename method refactorings following each other, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of RM_1 sets up one of the preconditions of RM_2: the method $C.M2$ must exist. The preconditions of the former refactoring are enough to guarantee the both transformations. We can replace both transformations by an alternative transformation. The new refactoring consists of the precondition of the former refactoring and the alternative transformation. $<RM_1 \Rightarrow RM_2> \equiv <RM_{new}>$, where $RM_{new_{C.M1 \Rightarrow C.M3}}$.

Example 4: There is a pair of refactorings $<EI, RC>$, where $EI_{C \Rightarrow I1}$, $RC_{I1 \Rightarrow I2}$. It means the "extract interface" refactoring is followed by "rename class" refactoring, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of EI sets up one of the preconditions of RC: the class $I1$ must exist. The preconditions of the former refactoring are enough to guarantee the both transformations. We can replace both transformations by an alternative transformation. The new refactoring consists of the precondition of the former refactoring and the alternative transformation. $<EI \Rightarrow RC> \equiv <EI>$, where $EI_{new_{C \Rightarrow I2}}$.

Example 5: There is a pair of refactorings $< RM, IM >$, where $RM1_{C.M1 \Rightarrow C.M2}$, $RM2_{C.M2 \Rightarrow \ldots}$. It means the "rename method" refactoring is followed by "inline method" refactoring, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of RM_1 sets up one of the preconditions of IM: the method $C.M2$ must exist. The preconditions of the former refactoring are enough to guarantee the both transformations. We can replace both transformations by an alternative transformation. The new refactoring consists of the precondition of the former refactoring and the alternative transformation. $<RM \Rightarrow IM> \equiv <IM>$, where $IM_{new_{C.M1 \Rightarrow \ldots}}$.

There are 37 predefined optimization rules. 4 rules are related to field-refactorings, 12 rules are related to method-refactorings, 11 rules are related to class-refactorings and 10 rules are related to mixed-type-refactorings. The above five examples of simplification rules have been integrated into our implementation. For other refactoring operators, the optimization rules of their operators are similar to these rules, so it is omitted here and can be referred to the table of this kind optimization rules in Appendix B.

2. Composition rules:

In general, for a pair of refactoring operations: $<RefOp_1$, $RefOp_2>$ in the chain of refactoring operations, if parts of the $PostKeys$ of the first refactoring are identical to parts of the $PreKeys$ of the second refactoring and there is the combination of the transformations by eliminating the preconditions of the second refactoring, we can apply the composition rules to the pair of refactorings.

The graphical depiction for composition rules is illustrated in Figure 3.6. The former refactoring's transformation can set up parts of the satisfaction of the latter refactoring's preconditions. We can eliminate the cost of checking preconditions of both refactorings[Rob99]. Based on the satisfaction of preconditions, we can compose the transformations for both refactorings.

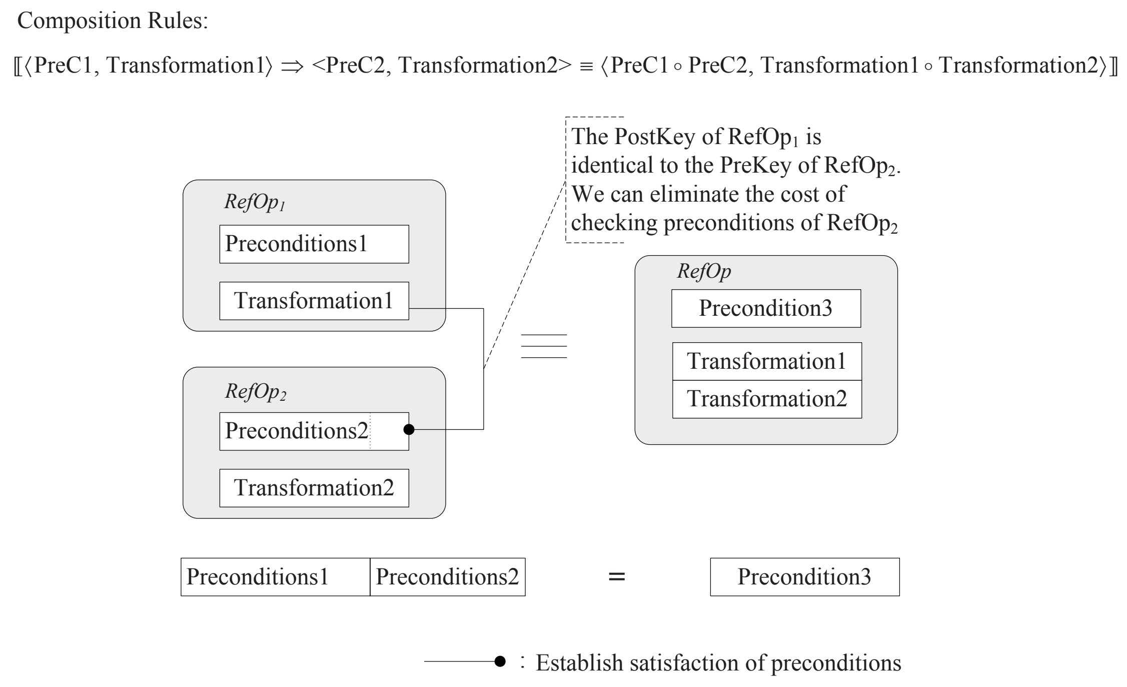

Composition Rules:
⟦⟨PreC1, Transformation1⟩ ⇒ <PreC2, Transformation2> ≡ ⟨PreC1 ∘ PreC2, Transformation1 ∘ Transformation2⟩⟧
The PostKey of RefOp1 is identical to the PreKey of RefOp2. We can eliminate the cost of checking preconditions of RefOp2
RefOp1
Preconditions1
Transformation1
RefOp2
Preconditions2
Transformation2
RefOp
Precondition3
Transformation1
Transformation2
Preconditions1
Preconditions2
=
Precondition3
: Establish satisfaction of preconditions

Figure 3.6: Unified composition rules

Example 1: There is a pair of refactorings $< RF, EF >$, where $RF_{C.f1 \Rightarrow C.f2}$, $EF_{C.f2 \Rightarrow C.f2|C.setf2|C.getf2}$. It means the "encapsulated field" refactoring is followed by "rename field" refactoring, if the transformation of the former refactoring sets up the parts of the preconditions of the latter refactoring. In this example, the transformation of RF sets up one of preconditions of EF: the field $C.f2$ must exist. The preconditions of the latter refactoring to be checked are eliminated. We can compose both transformations together and merge both preconditions to eliminate the cost of checking preconditions. The new refactoring consists of the merged preconditions and the composed transformations. $<RF \Rightarrow EF> \equiv <RF \circ EF>$.

Example 2: There is a pair of refactorings $<MM, RM>$, where $MM_{C1.M|C2 \Rightarrow C1|C2.M}$, $RM_{C2.M \Rightarrow C2.M2}$. It means the "move method" refactoring is followed by "rename method" refactoring, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of MM sets up one of the preconditions of RM: the method $C2.M$ must exist.

The preconditions of the latter refactoring to be checked are eliminated. We can compose both transformations together and merge the both preconditions to eliminate the cost of checking preconditions. The new refactoring consists of the merged preconditions and the composed transformations. $<MM \Rightarrow RM> \equiv <MM \circ RM>$.

Example 3: There is a pair of refactorings $<MM, SA>$, where $MM_{C1.M|C2 \Rightarrow C1|C2.M}$, $SA_{C2.M \Rightarrow C2.M}$. It means the "move method" refactoring is followed by "Substitute Algorithm" refactoring, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of MM sets up one of the preconditions of $C2.M$: the method $C2.M$ must exist. The preconditions of the latter refactoring to be checked are eliminated. We can compose both transformations together and merge the both preconditions to eliminate the cost of checking preconditions. The new refactoring consists of the merged preconditions and the composed transformations. $<MM \Rightarrow SA> \equiv <MM \circ SA>$.

Example 4: There is a pair of refactorings $< SA, RM >$, where $SA_{C.M1 \Rightarrow C.M1}$, $RM_{C.M1 \Rightarrow C.M2}$. It means the "substitute algorithm" refactoring is followed by "rename method" refactoring, if the transformation of the former refactoring sets up the parts of preconditions of the latter refactoring. In this example, the transformation of SA sets up one of the preconditions of RM: the method $C.M1$ must exist. The preconditions of the latter refactoring to be checked are eliminated. We can compose both transformations together and merge the both preconditions to eliminate the cost of checking preconditions. The new refactoring consists of the merged preconditions and the composed transformations. $<SA \Rightarrow RM> \equiv <SA \circ RM>$.

The composition of refactorings is widely researched[Rob99][Kni04][CN00]. If it necessary, we could predefines the composition rules for the pairs of standard refactorings and don't consider more than two standard refactorings, because the number of refactorings can be composed is infinite. The intention of composing a pair of refactorings to a non-standard refactorings is to eliminate the cost of preconditions checking based on user requirements.

3.4 Optimization

We use the term *compose plan* of RFMs to denote the configuration of the user-defined sequence of refactorings. Applying the user-defined sequence of these refactoring operations is equivalent to executing the *compose plan* of corresponding RFMs. Therefore, optimizing this sequence essentially can be seen as finding a better *compose plan* for corresponding RFMs. The process of optimization takes place in two separate phases: the first phase is called logical optimization and the second called physical optimization. Both phases follow on the translation step as listed below:

- 1. Translate *compose plan* into its formal expression for the sequence of refactoring operations.

- 2. Perform logical optimization

- 3. Perform physical optimization

3.4.1 Logical Optimization

Theoretically, the foundation for logical optimization is formed by the set of formal equivalent expressions of a given user-defined sequence. Given an initial formal expression, which results from the translation of the given user-defined sequence, the optimization rules mentioned in the preceding section can be used to derive some formal expressions that are equivalent to the initial algebraic expression. The set of formal equivalent expressions spans the possible optimized sequences for a user-defined original sequence.

In the logical level, the optimization focuses on two steps:

- Adjust locations of refactoring operations in initial sequence. This step exposes the potential commutative relationships and dependency relationships. Based on the different relationships among refactoring operations, the initial sequence is split into different chains of refactoring operations that expose optimization potentials by reordering locations of refactoring operations.
- Apply optimization rules to the initial sequence. If the initial sequence is an disadvantageous sequence, we could achieve an optimized sequence for a user-defined original sequence.

To illustrate, There is a sequence of refactorings defined in RFMs and the compose plan of RFMs as follows:

1. "RenameClass" refactoring: rename class name $C1$ to $C2$,
2. "RenameMethod" refactoring: rename method name $M1$ in $C2$ to $M2$
3. "RenameField" refactoring: rename field name $F1$ in $C2$ to $F2$,
4. "RenameMethod" refactoring: rename method name $M2$ in $C2$ to $M3$,
5. "RenameClass" refactoring: rename class name $C2$ to $C3$,
6. "RenameMethod" refactoring: rename method name $M3$ in $C3$ to $M4$,

As a legal sequence, the preconditions of these refactorings have already satisfied. Based on the identifiers for detailed program elements, we derived the $PreKey$ and $PostKey$ for the six corresponding refactoring operations. A user-defined sequence of refactoring operations $< RC_1, RM_1, RF_1, RM_2, RM_3, RC_2 >$ is illustrated in Figure 3.7, where

1. RC_1 is "RenameClass" refactoring: $RC1_{C1 \Rightarrow C2}$,
2. RM_1 is "RenameMethod" refactoring: $RM1_{C2.M1 \Rightarrow C2.M2}$,
3. RF_1 is "RenameField" refactoring: $RF1_{C2.F1 \Rightarrow C2.F2}$,
4. RM_2 is "RenameMethod" refactoring: $RM2_{C2.M2 \Rightarrow C2.M3}$,
5. RC_2 is "RenameClass" refactoring: $RC2_{C2 \Rightarrow C3}$,
6. RM_3 is "RenameMethod" refactoring: $RM3_{C3.M3 \Rightarrow C3.M4}$,

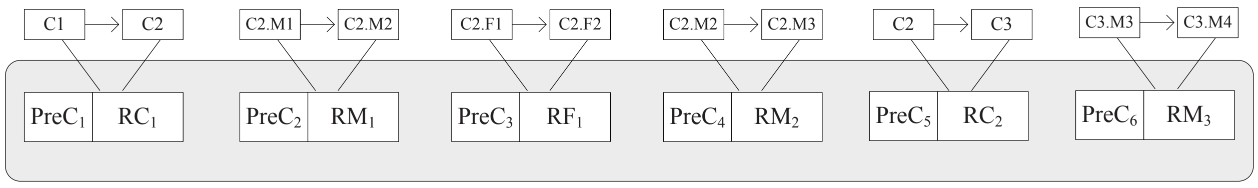


Figure 3.7: The initial sequence of six refactoring operations

As illustrated in Figure 3.7, this sequence has six refactoring operations and can be executed successfully in the left to right order.

To optimize this sequence, we firstly explore relationships among refactoring operations, especially dependency relationships which are graphically shown in Figure 3.8. If there is a dependency relationship between any two refactoring operations, we should keep the order of these refactorings. In the dependency relationship, we further explore the optimization chance by matching predefined optimization rules. In Figure 3.8, there are conditional commutative operations between RC_2 and RM_2, RC_2 and RF_1, RC_2 and RM_1. RC_2 does not depend on RM_2, RF_1 and RM_1. But the identifier of $PreKey$ and $PostKey$ of RM_2, RF_1 and RM_1 depend on the identifier of $PreKey$ of RC_1. Thus, after applying optimization rules to RC_1 and RC_2, we should update the $PreKey$ and the $PostKey$ of RM_2, RF_1 and RM_1 to guarantee the legal execution of the optimized sequence.

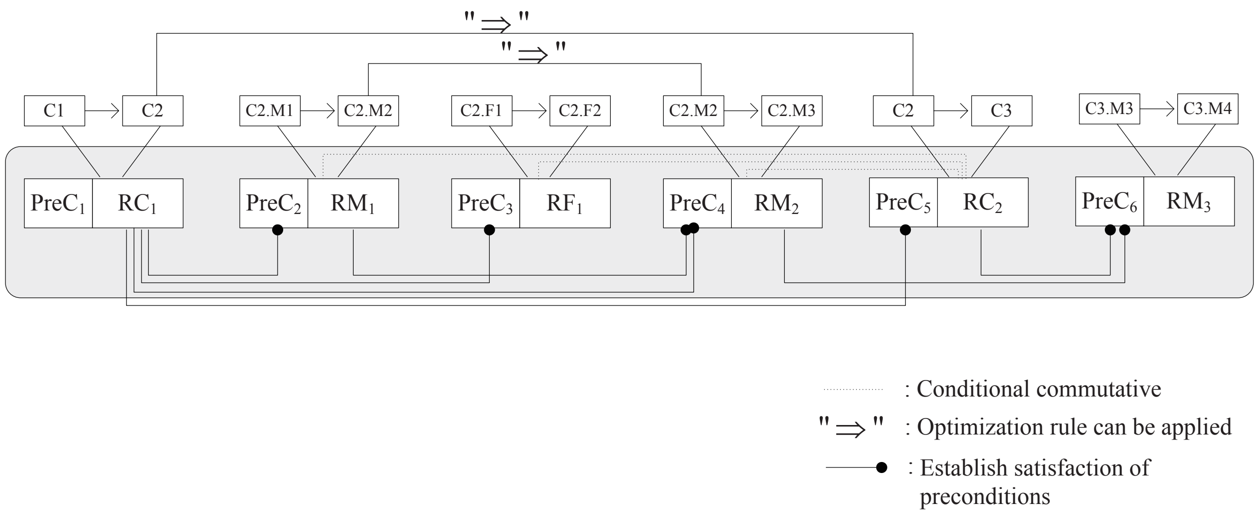

"⇒"
"⇒"
C1
C2
C2.M1
C2.M2
C2.F1
C2.F2
C2.M2
C2.M3
C2
C3
C3.M3
C3.M4
PreC1
RC1
PreC2
RM1
PreC3
RF1
PreC4
RM2
PreC5
RC2
PreC6
RM3
: Conditional commutative
"⇒" : Optimization rule can be applied
: Establish satisfaction of preconditions

Figure 3.8: Potential the relationships in initial sequence

Secondly, as illustrated in Figure 3.9, refactoring operations in the initial sequence are separated into different chains by reordering their locations. After we updated the the $PreKey$ and the $PostKey$ of RM_2, RF_1 and RM_1, the new optimization chance is also exposed between RM_2 and RM_3. For each chain of refactoring operations, the relational operator "$\rightarrow$" and "$\Rightarrow$" are used to process the pairs of refactorings in an iterative manner in our concept. If there is an optimization rule that can be applied to the pair of refactorings, we use the relational operator "$\Rightarrow$" to express that this pair of refactorings can be optimized. If not, we should keep the order of the two refactorings.

Thirdly, after we apply optimization rules to different chains of refactoring opera-

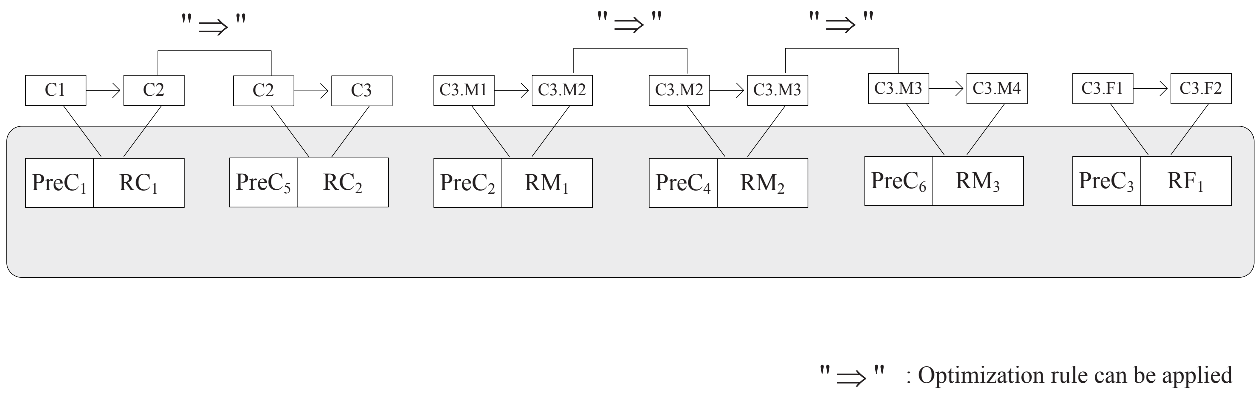


Figure 3.9: Reorder sequence for applying optimization rules

tions, the optimized sequences can be generated. The applied optimization rules contain $[RC1_{C1\Rightarrow C2} \Rightarrow RC2_{C2\Rightarrow C3} \equiv RCnew_{C1\Rightarrow C3}$, where the $PostKey$ of RC_1= $PreKey$ of $RC_2]$, $[RM1_{C3.M1\Rightarrow C3.M2} \Rightarrow RM2_{C3.M2\Rightarrow C3.M3} \equiv RMnew_{C3.M1\Rightarrow C3.M3}$, where the $PostKey$ of RM_1= $PreKey$ of $RM_2]$, $[RM1_{C3.M2\Rightarrow C3.M3} \Rightarrow RM2_{C3.M3\Rightarrow C3.M4} \equiv RMnew_{C3.M2\Rightarrow C3.M4}$, where the $PostKey$ of RM_1= $PreKey$ of $RM_2]$, where the $PostKey$ of RF_1= $PreKey$ of $RF_2]$. They belong to one kind of the optimization rules, namely simplification rules that simply the two refactorings to one standard refactoring. After we apply this kind of optimization rules, we usually can get the best optimization effect for the applied sequence. For example in Figure 3.10, the $<RC_{new}, RM_{new}, RF_1>$ is the best optimized sequence. Its compilation time should be less than any other equivalent sequences whose compilation results are the same.

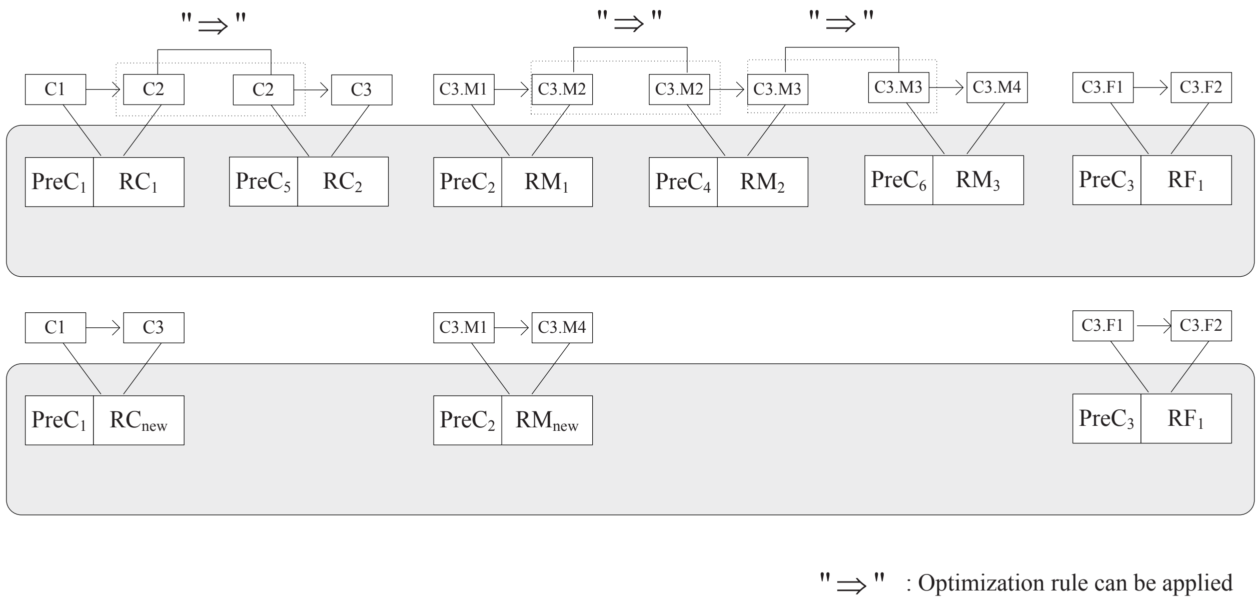


Figure 3.10: Apply optimization rules

Another kind of optimization rules is the composition rule that composes two operations together. In an extreme situation, if each chain of refactorings can be composed to a composite refactoring, this sequence could be the best optimized sequence. But it is difficult to implement this in practical application, because our concepts only predefine the composition rules for the pair of refactorings. For example, there is a three-refactoring-operation chain $<RF, EF, IM>$ where $RF_{C.f1 \Rightarrow C.f2}$, $EF_{C.f2 \Rightarrow C.f2|C.setf2|C.getf2}$, $IM_{C.getf2 \Rightarrow ...}$. We have to choose one pair of refactorings between $<RF, EF>$ and $<EF, IM>$ to be the optimized by applying optimization rules. At this moment, the concrete program should be analyzed. The concept of physical optimization is represented to determine which pair of refactorings should be applied optimization rules in the rest of this chapter.

3.4.2 Physical Optimization

Physical optimization is a complement concept to support logical optimization. It guarantees the optimized sequence of refactorings to be a legal sequence and gets better optimization effect by considering concrete programs.

On one hand, when we reorder an initial sequence in practice, reordering the initial sequence of these refactorings need to analyze the concrete program. Otherwise it could break potential dependencies and influence the satisfaction of precondition of the involved refactoring. For example, there are class $C1$ and $C1$'s subclass $C2$. In the class $C2$, the method $C2.M1$ overrides $C1.M1$. Due to the semantics of Java language, when a legal "rename method" refactoring renames the $C1.M1$ into $C1.M2$, the $C2.M1$ is also renamed to $C2.M2$. Given a three-refactoring-sequence $<RM_1, SA^5, RM_2>$ where ${RM_1}_{C1.M1 \Rightarrow C1.M2}$, $SA_{C2.M2 \Rightarrow C2.M2}$, ${RM_2}_{C1.M2 \Rightarrow C1.M3}$, we could consider the RM_1 and RM_2 to expose optimization potential without analyzing the concrete program. In fact, changing locations of SA and RM_2 breaks the satisfaction of precondition of SA. The solution of this kind of potential dependencies is that we introduce additional identifiers from the concrete program. These identifiers are also transformed to establish preconditions for some later refactorings. In this example, the additional identifier of RM_1's $PreKey$ is "$C2.M1$" and is transformed to "$C2.M2$". "$C2.M2$" to exist is one of the precondition of SA where $SA_{C2.M2 \Rightarrow C2.M2}$. It can not be broken. In other words, the locations of SA and RM_2 can not be reordered.

On the other hand, if more than two types of refactorings expose optimization potential, we could apply composition rules to pairs of refactorings. For example, there is a three-refactoring chain $<RC, RM, MM^6>$ where $RC_{C1 \Rightarrow C2}$, $RM_{C.M \Rightarrow C.newM}$ and $MM_{C.newM|C2 \Rightarrow C|C2.newM}$. In an ideal situation, these refactorings can be merged together. But due to limitation of our concept, we predefine composition rules for composing two refactorings manually. Hence, we should choose one better optimization potential from $RM \circ MM$ and $RC \circ MM$. The solution to this limitation is that we choose one of identifiers as $PreKey$ or $PostKey$ to expose optimization potential, while other relevant identifiers as additional identifiers. These identifiers also need to be guaranteed by keeping the order of refactorings and unoptimized refactorings. The choice of identifier as $PreKey$ or $PostKey$ could require to analyze concrete program. The cost of relevant identifiers checking often dominates the cost of the refactorings[KK04]. Because the program elements operated by refactorings are local whereas the related preconditions checking is global. For example, "Remove Setting Method" refactoring needs a constant

[5]SA denote the "substitute algorithm" refactoring which replaces the method body of $C2.M2$ with new algorithm

[6]MM denote "move method" refactoring which moves the method $newM$ from class C into class $C2$

time and only locally deletes corresponding methods, whereas checking whether it is legal requires searching more detailed elements in the program. In our concept, the optimization effects of composition rules of pairs of refactorings are based on eliminating the process of checking preconditions and merging the transformations. Therefore, the physical optimization focuses on estimating the cost of checking the detailed elements. The identifiers are estimated based on the statistics about cost of checking identifiers in the concrete program. The more positions of an identifier existing in the concrete program, the higher the estimated cost of the identifier. We should compose preconditions involving the identifier with higher estimated cost.

To illustrate, we introduce the concept of physical optimization about applying composition rules in the following three general situations. At the same time, we come up with the corresponding solutions to compromise optimizations for guaranteeing the execution of optimized sequence as follows:

Situation 1:

In this situation, only one identifier in the $PreKey$ and the $PostKey$ of involved refactorings expose the optimization potential. We take the general three-refactoring sequence for instance illustrated in Figure 3.11. According to our opti-

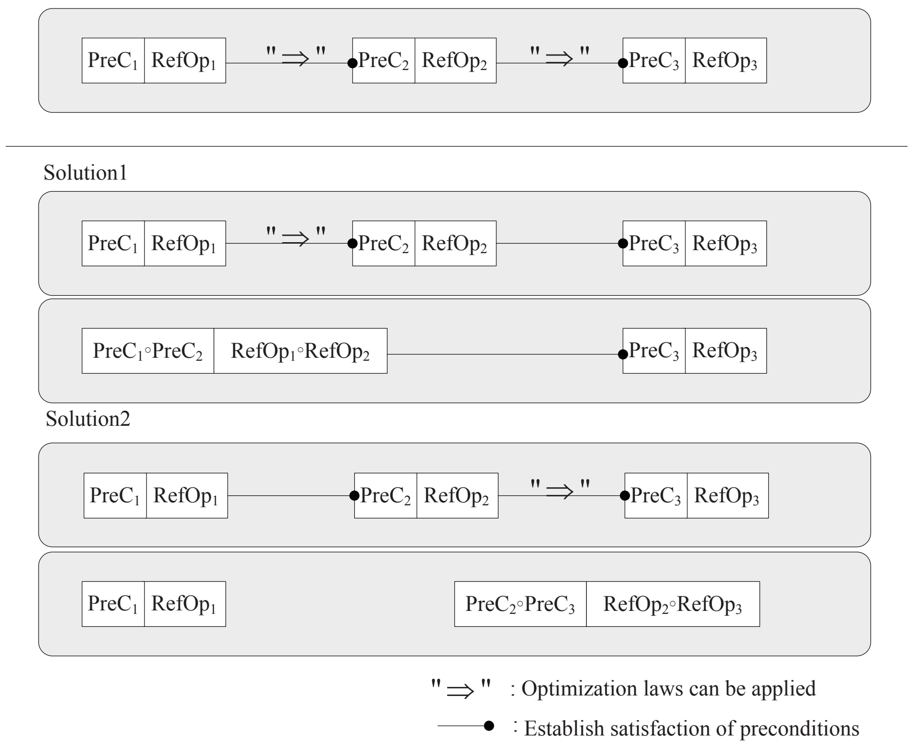

PreC₁ RefOp₁
"⇒"
PreC₂ RefOp₂
"⇒"
PreC₃ RefOp₃
Solution1
PreC₁∘PreC₂ RefOp₁∘RefOp₂
Solution2
PreC₂∘PreC₃ RefOp₂∘RefOp₃
"⇒" : Optimization laws can be applied
: Establish satisfaction of preconditions

Figure 3.11: Situation 1

mization rules, especially the type of rules belongs to composition rules, are applied in the pair of refactorings. We only consider the combination of two transformations. Although there is a potential chance to compose the three transformations in the ideal situation, we only focus on two transformations. The rest one keeps its position in this sequence. Take a three-refactoring-operation chain $<SA, RM, MM>$ for example, where $SA_{C.M \Rightarrow C.M}$, $RM_{C.M \Rightarrow C.newM}$ and $MM_{C.newM|C2 \Rightarrow C|C2.newM}$. In ideal situation, we can compose a three-refactoring operation $<SA \circ RM \circ MM>$. But our concept only supports to merge the pair of refactoring operations. Hence, we have to choose the better optimization between the $<SA \circ RM, MM>$ and the $<SA, RM \circ MM>$. By using this way, the optimized sequence can be guaranteed to execute legally. In this example, the cost of checking identifier "$C.M$" is identical to identifier "$C.newM$". So the optimization effect of $<SA \circ RM, MM>$ is the same as that of $<SA, RM \circ MM>$. We can apply composition rules for $<SA \circ RM, MM>$ or $<SA, RM \circ MM>$.

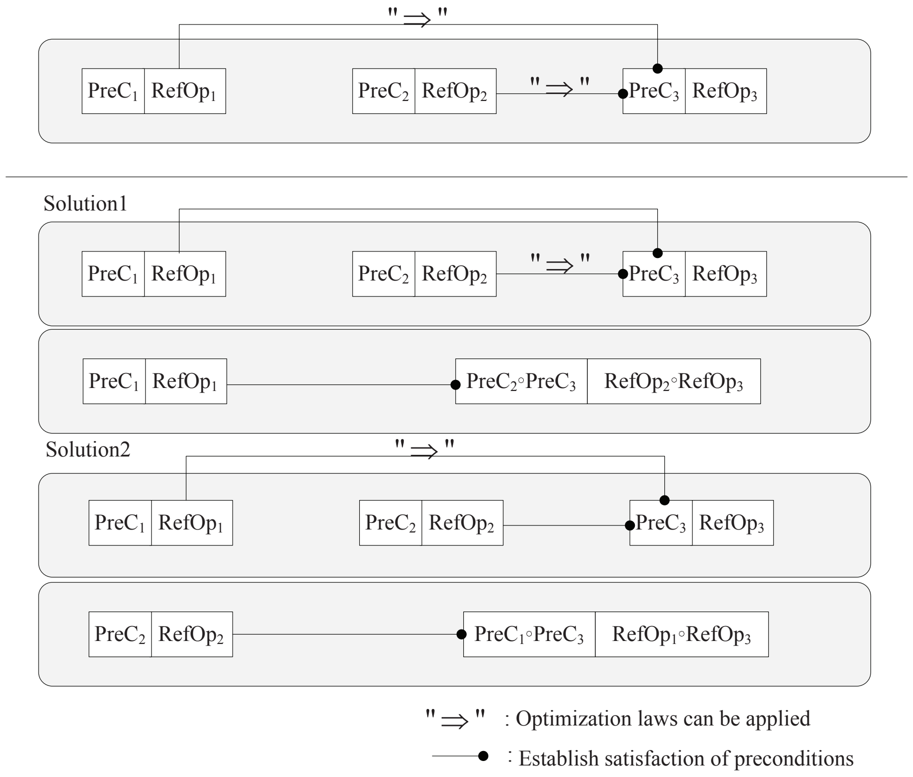

"⇒"
PreC1 RefOp1
PreC2 RefOp2
"⇒"
PreC3 RefOp3
Solution1
PreC2◦PreC3 RefOp2◦RefOp3
Solution2
PreC1◦PreC3 RefOp1◦RefOp3
"⇒" : Optimization laws can be applied
: Establish satisfaction of preconditions

Figure 3.12: Situation 2

Situation 2:

In this situation, more than one identifier in the $PreKey$ of involved refactorings expose the optimization chances. As illustrated in Figure 3.12, we should choose one better solution between solution1 and solution2. We take the general three-refactoring sequence for instance illustrated in Figure 3.12.

For the same reason in Situation 1, we need to optimize the sequence and guarantee that it is executed legally. For example, there is a three-refactoring-operation chain $<RC, RM, MM>$ where $RC_{C1 \Rightarrow C2}$, $RM_{C.M \Rightarrow C.newM}$ and $MM_{C.newM|C2 \Rightarrow C|C2.newM}$. In ideal situation, we can compose a three-refactoring operation $<RC \circ RM \circ MM>$. But our concept only supports to merge the pair of refactoring operations. Hence, we have to choose the better optimization between the $<RC, RM \circ MM>$ and the $<RM, RC \circ MM>$. Which one is better optimization sequence is based on the statistics about the cost of checking identifier "$C2$" and identifier "$C.newM$" in the concrete program. The more positions of identifier need to be checked, the higher estimated cost of the identifier. The optimization effect is better by eliminating the larger cost of checking preconditions. By keeping the order of optimized refactoring and unoptimized refactoring, the optimized sequence also can be guaranteed to execute legally.

Situation 3:

In this situation, more than one identifier in the $PostKey$ of involved refactorings expose the optimization chances. As illustrated in Figure 3.13, we should choose one better solution between solution1 and solution2. We take the general three-refactoring sequence for example illustrated in Figure 3.13. For the same reason in Situation 2, we consider optimizing the sequence and guarantee it is executed legally. For example, there is a three-refactoring-operation chain $<EF, IM, RF>$ where $EF_{C.f \Rightarrow C.f|C.getf|C.setf}$, $IM_{C.get \Rightarrow ...}$ and $RF_{C.f| \Rightarrow C.newf}$. In an ideal situation, we can compose three refactoring operation $< EF \circ IM \circ RF >$. But our concept only supports to merge the pair of refactoring operations. Hence, we have to choose the better optimization between the $<EF \circ IM, RF>$ and the $<EF \circ RF, IM>$. Which one is the better optimization sequence is based on the statistics about the cost of checking identifiers "$C.f$" and identifier "$C.getf$" in concrete program. In this example, due to the fact that the field are encapsulated and that accessing field depends on the "$C.getf$", the $< EF \circ IM, RF >$ could be better sequence.

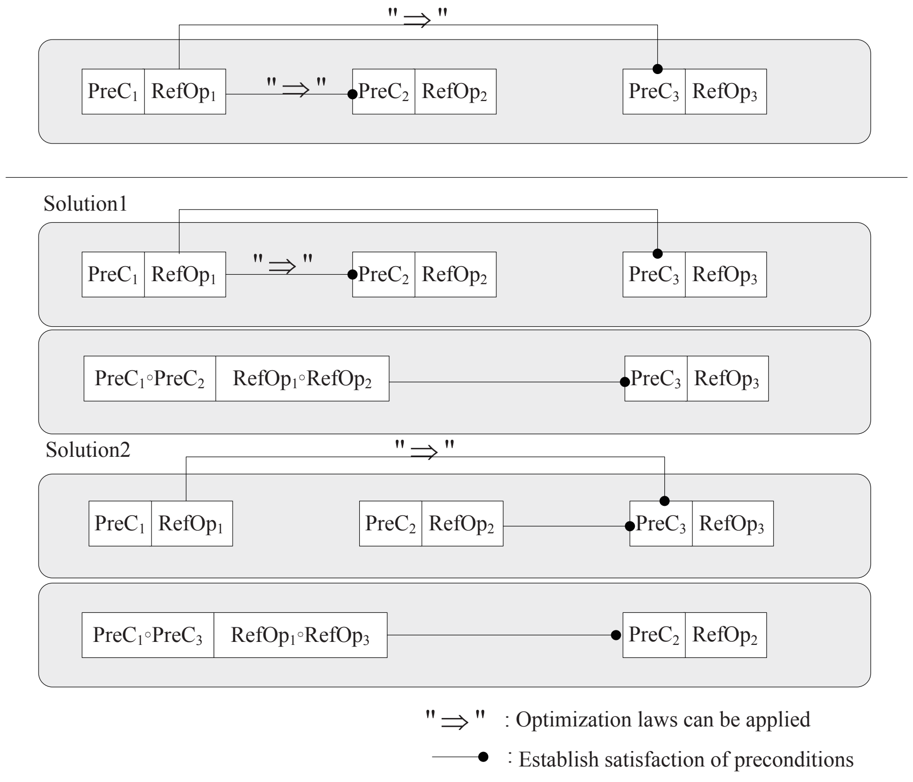

"⇒"
PreC1 RefOp1
"⇒"
PreC2 RefOp2
PreC3 RefOp3
Solution1
PreC1∘PreC2 RefOp1∘RefOp2
Solution2
PreC1∘PreC3 RefOp1∘RefOp3
"⇒" : Optimization laws can be applied
: Establish satisfaction of preconditions

Figure 3.13: Situation 3

To sum up, as we take account of the concrete program, the optimizing sequences of refactorings are not easy to implement automatically in physical level. Realistically, in the step of reordering sequence of logical level, reordering the sequence needs to consider more preconditions checking. The satisfaction of these conditions could consider to analyze the whole concrete program and preconditions of different types of refactorings. In practical application, we need more techniques to detect the more complex relationships among specified refactorings based on the concrete program[MTR06]. When applying optimization rules in steps of logical level, if it is necessary, the physical optimization is a good complement to support finding better optimization sequence by statistics about the detailed program elements.

Chapter 4

Implementation of Prototype

The concepts of the logical optimization of user-defined sequences of refactorings in RFMs described so far are implemented in a prototype called *RFMoptimizer*. *RFMoptimizer* is a program for optimizing sequences of RFMs. Its application can be demonstrated by using it as an additional program for an existing compose tool (*RFMcomposer*). If sequences can be optimized, it can generate the optimized sequence of RFMs instead of the original sequence of RFMs before *RFMcomposer* composes these RFMs.

In this chapter, we firstly introduce the functional requirements analysis of the ideal *RFMoptimizer* in Section 4.1. Moreover, based on the functional requirements analysis, we design a prototype for the *RFMoptimizer* to demonstrate the optimization's effects in Section 4.2. Finally, we discuss the limitations in the application of this prototype in Section 4.3.

4.1 The Functional Requirements Analysis

In the preceding chapter, we describe the concepts of optimizing sequences of refactorings. In practical application, as these refactorings are encapsulated in the RFMs, our implementation focuses on optimizing the composition of RFMs. Figure 4.1 describes the role of our *RFMoptimizer* in the whole of composing RFMs. The importance of our optimization is to generate optimized sequences of RFMs before RFMs are composed. An ideal *RFMoptimizer* for optimizing sequences of refactorings in RFMs consists of four main functions as listed below:

loading refactorings:

This function focuses on loading the user-defined sequence of RFMs. First of all, it loads the specified refactorings defined in a equation file[1]. Then it extracts each individual refactoring from the corresponding refactoring units encapsulated in RFMs. Lastly, it produces the sequence of specified refactorings. The implementation of loading RFMs has been reused in the existing tool RFMcomposer[KBA08]. The extension of our implementation only requires recording the corresponding sequence of refactorings from loading RFMs.

[1]The equation file is a configuration file for existing compose tool, which includes the user-selected RFMs and execution order of these RFMs.

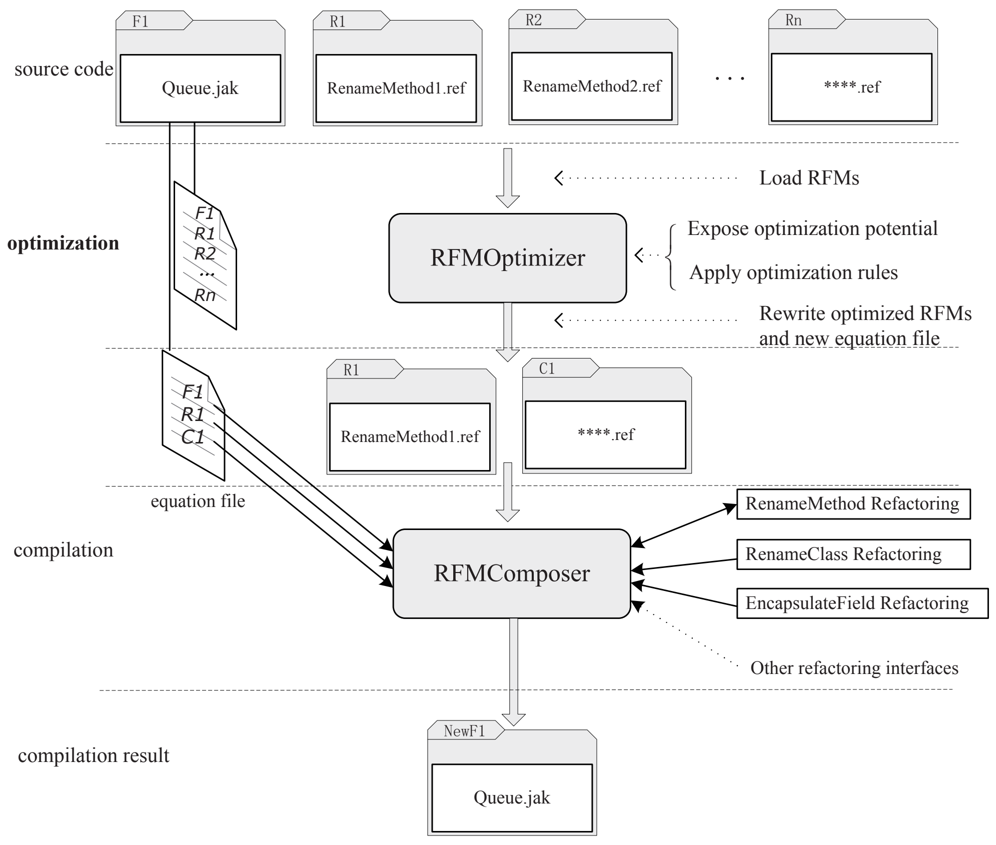


Figure 4.1: The role of optimization process in composition of RFMs

reordering refactorings:

This function focuses on processing the sequence of refactorings in initial order. The intention of reordering refactorings is to expose optimization potential among relevant refactorings. Firstly it checks the potential commutative and dependency relationships among refactorings. To illustrate, there is a five-refactoring sequence $< RF_1, RM_1, RC_1, RM_2, RC_2 >$, where $RF_{1_{C1.f1 \Rightarrow C1.f2}}$, $RM_{1_{C9.M1 \Rightarrow C9.M2}}$, $RC_{1_{C1 \Rightarrow C2}}$, $RM_{2_{C9.M2 \Rightarrow C9.M3}}$, $RC_{2_{C2 \Rightarrow C3}}$. In this sequence, the RM_2 depends on the RM_1, the RC_2 depends on the RC_1 and there are commutative relationships among other refactorings. Secondly it separates the sequence into different chains of refactorings[2]. We can get three chains of refactorings in the previous example, i.e., $<RF_1>$, $<RC_1, RC_2>$, $<RM_1, RM_2>$. However, sometimes we need to reorder the location of refactorings in more complex situations. For example, in our theory, we mentioned the set-up-part-of-identifier dependency, there is three-refactoring sequence $< RC_1, RM, RC_2 >$, where $RC_{1_{C1 \Rightarrow C2}}$, $RM_{C2.M1 \Rightarrow C2.M2}$, $RC_{2_{C2 \Rightarrow C3}}$. We usually swap the locations of RM and RC_2, because there is optimization potential between RC_1 and RC_2. But the $PreKey$ and the $PostKey$ of RM depends on RC_1. We should firstly update the $PreKey$ and the $PostKey$ of RM based on RC_2. Therefore, in this situation, we need to update the corresponding parameters for relevant refactorings. After the above example is reordered, the updated three-refactoring

[2]Each chain of refactorings corresponds to a subsequence in which all refactorings depend on each other

sequence is $< RC_1, RC_2, RM >$, where $RC1_{C1\Rightarrow C2}$, $RC2_{C2\Rightarrow C3}$ and $RM_{C3.M1\Rightarrow C3.M2}$. In practical application, it could be more complex that we reorder the sequence to expose the chain of refactorings. Finally, the sequence of refactorings is reordered to expose optimization chances and guaranteed as a legal sequence.

applying optimization rules:

The updated sequence of refactorings that has already been exposed optimization potentials is generated from the above function. This function focuses on optimizing this sequence. Firstly, it analyzes the pairs of refactorings within these chains one by one. Secondly, if the pairs of refactorings exposed the chance of optimization, it applies optimization rules to the pairs of refactorings by matching the predefined optimization rules. If not, we should keep the order of the refactorings. In practical application, take the chain of refactorings $<RC_1, RC_2, RM>$ for example, where $RC1_{C1\Rightarrow C2}$, $RC2_{C2\Rightarrow C3}$ and $RM_{C3.M1\Rightarrow C3.M2}$. it optimizes the pair of refactorings RC_1 and RC_2 by matching the optimization rule $[RC1_{C1\Rightarrow C2} \Rightarrow RC2_{C2\Rightarrow C3} \equiv RC_{new_{C1\Rightarrow C3}}$, where the $PostKey$ of RC_1 = the $PreKey$ of $RC_2]$. Lastly, the optimized sequences of refactorings is generated. For above example, after optimization rule is applied, the updated sequence is $<RC_{new}, RM>$ where $RC_{new_{C1\Rightarrow C3}}$ and $RM_{C3.M1\Rightarrow C3.M2}$

rewriting refactorings and the order of execution:

This function focuses on reflecting the information of optimized sequences of refactorings into *RFMcomposer*. For one thing, it rewrites the updated refactorings into the corresponding RFMs. For another, the equation file is rewritten to update the order of the composition of the updated RFMs.

The four functions are the basis of the implementation of an ideal *RFMoptimizer*. In order to verify the optimization effect for composition of RFMs in practice, we design one prototype and implement one demo program *RFMoptimizer* for optimizing sequences of RFMs. In the following section, we will describe the framework of Prototype.

4.2 The Design of Prototype

4.2.1 The Framework of Prototype

The framework of prototype consists of four modules to implement the four functions in the previous section. It includes a refactorings loading module, a refactorings reordering module, an optimization rules applying module, an equation file and updated RFMs rewriting module. In the prototype, we integrate these concepts into a program without influencing the work of existing compose tool. Therefore, we design a simplified program *RFMoptimizer* which is independent of the *RFMcomposer*.

Figure 4.2 depicts a simplified UML for this *RFMoptimizer*. First of all, the class *UnifiedRefactoring* is used to unify the different types of refactorings. Although different refactorings are implemented in various ways, different types of refactorings have several specified and unified information required to be extracted. In general, this information

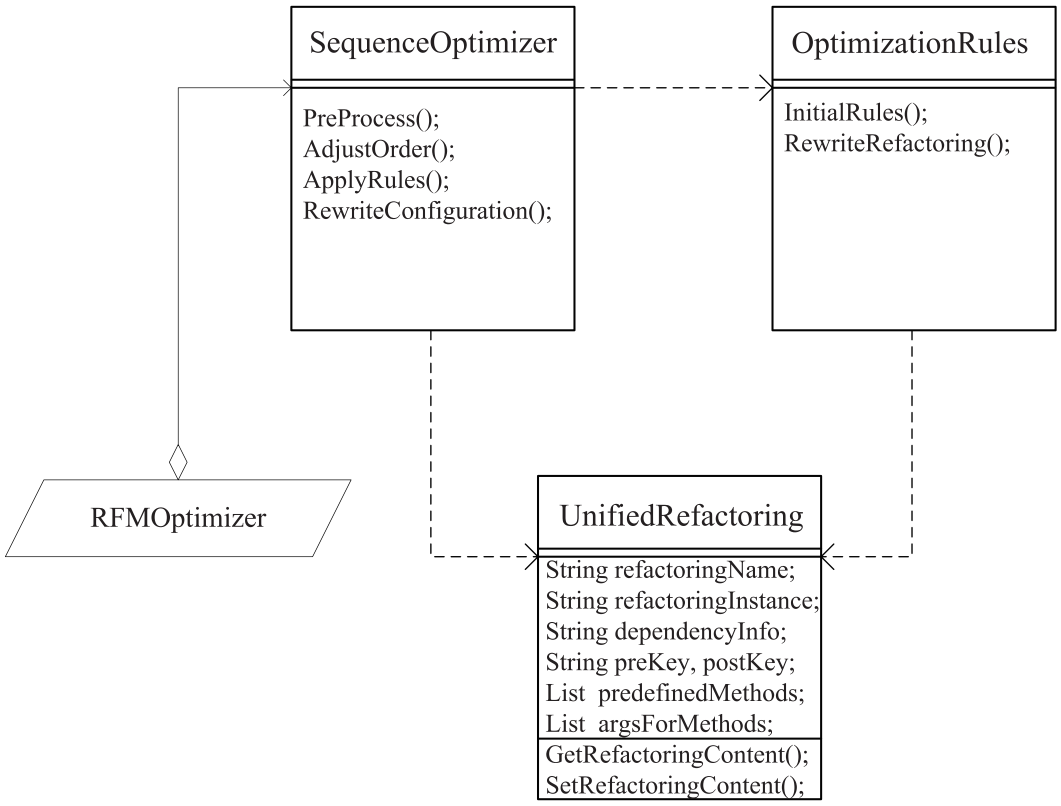


Figure 4.2: Simplified UML of the core classes in Prototype

includes a refactoring name [3] and a refactoring instance name [4], the *PreKey* and the *PostKey*, getter/setter methods and corresponding parameters, a dependency description[5], a chain information[6] and so on. For example, there is a three-refactoring sequence $< RC_1, RM, RC_2 >$, $RC_1 := (C1, C2)$, $RM := (C2.M1, C2.M2)$, $RC_2 := (C2, C3)$. In practical application, the three specified refactorings involve two types of refactorings, namely the "Rename Class" refactoring and the "Rename Method" refactoring. RC_1 and RC_2 belong to the "Rename Class" refactoring; RM belongs to the "Rename Method" refactoring. Before optimization process, the intention of the class *UnifiedRefactoring* provides a template that collects the unified information from types of refactorings and acquires the specified information from the specified refactorings. For RC_1, we can capture the $PreKey$ is $C1$, the $PostKey$ is $C2$ and the dependency description is that the RM depend on it. Besides, we define the optimization rules for a number of pairs of refactorings in the class *OptimizationRules* based on the refactoring instances and corresponding *PreKey* and *PostKey*. For example, in the class *OptimizationRules*, we can define the optimization rules such as $[RC_{1_{C1\Rightarrow C2}} \Rightarrow RC_{2_{C2\Rightarrow C3}} \equiv RC_{new_{C1\Rightarrow C3}}]$, $[RM_{1_{C.M1\Rightarrow C.M2}} \Rightarrow RM_{2_{C.M2\Rightarrow C.M3}} \equiv RM_{new_{C.M1\Rightarrow C.M3}}]$ and $[RF_{1_{C.f1\Rightarrow C.f2}} \Rightarrow RF_{2_{C.f2\Rightarrow C.f3}} \equiv RF_{new_{C.f1\Rightarrow C.f3}}]$. Last but not least, we implement the main functions in the class *SequenceOptimizer*

[3] A refactoring name corresponds to the type name of refactoring.

[4] A refactoring instance name corresponds to the name of the specified refactoring.

[5] The dependency description of this specified refactoring can record the list of other specified refactorings that depend on it.

[6] Chain information can track the original program elements by recording the track of corresponding refactoring operations.

by associating with the class *OptimizationRules* the class *UnifiedRefactoring*. In the following section, we will introduce the core algorithms in the main functions.

4.2.2 The Core Algorithms

The class *OptimizationRules* and the class *UnifiedRefactorings* are auxiliary classes. The implementation of both classes is not complicated, so it is omitted here and can be referred to the source code. In fact, *SequenceOptimizer* is the main class of our optimization tool. Therefore, in this section, we mainly introduce two core algorithms that are applied for the reordering refactorings and the applying optimization rules function in the class *SequenceOptimizer*.

Algorithm 1 Optimizing the order of the initial sequence

```
Input:
    The initial sequence of refactorings($SEQ$: $< R_1, R_2, ...R_n >$).
Output:
    The new order sequence of refactorings($SEQ'$),where $R$s with updated arguments.
 1: Set analyzedIdx and unanalyzedIdx for the position of $R$s in $SEQ$
 2: for $unanalyzedIdx = 1$; $unanalyzedIdx < SEQ.length$; $unanalyzedIdx++$ do
 3:   Let $analyzedIdx = unanalyzedIdx - 1$;
 4:   for $i = unanalyzedIdx$; $i < SEQ.length$; $i++$ do
 5:     if The $PostKey$ of the $R_{analyzedIdx}$ == The $PreKey$ of the $R_i$ then
 6:       Detect the relationship between $R_i$ and $R$;{$R$ are iteratively from
          $R_{unanalyzedIdx}$ to $R_i$}. If the relationship can not be broken, then break;
 7:       else: Move the $R_i$ into the position $unanalyzedIdx$; {adjust the order of SEQ}
 8:     else if The $PreKey$ of $R_i$ depends on the $PostKey$ of $R_{analyzedIdx}$ then
 9:       Add $R_i$ to the list of related refactorings for $R_{analyzedIdx}$;
10:     end if
11:   end for
12: end for
13: Let $SEQ' = SEQ$;
14: for $i = 0$; $i < SEQ'.length$; $i++$ do
15:   if $i < SEQ'.length - 1$ and $R_i \circ R_{i+1}$[7] and $PostKey$ of $R_i$ = $PreKey$ of $R_{i+1}$
      then
16:     if The list of $R_i$'s related refactorings is not empty then
17:       Update the arguments, $PreKey$ and $PostKey$ to related refactorings of $R_i$;
18:       if The list of $R_{i+1}$'s related refactorings is not empty then
19:         Add the list of related refactoring to $R_{i+1}$;
20:       else
21:         Set the list of related refactoring to $R_{i+1}$;
22:       end if
23:     end if
24:   end if
25: end for
```

[7]It checks the pair of refactorings whether can expose optimization potential. if it can, the $R_i \circ R_{i+1}$ is TRUE.

The first algorithm is a simplified solution to the problem of rearranging the locations of refactorings in the user-defined sequence. It is a preparation for applying optimization rules. The intention of reordering the locations of refactorings is to separate the original sequence into several chains of refactorings. As mentioned in the previous chapter, a chain of refactorings means that refactorings in the chain depend on each other. Therefore, the basic idea of reordering the refactorings is based on finding the potential dependency relationship. To simplify the practical application, we determine whether there is a dependency relationship between the two refactorings based on two situations:

- The *PostKey* of the former refactoring is identical to the *PreKey* of the latter refactoring.
- The *PostKey* of the former refactoring is part of the *PreKey* of the latter refactoring.

The process of reordering locations of refactorings is described as follows: The sequence of refactorings is split into two parts. The first part includes refactorings which have been analyzed. In this part, the positions of refactorings have already been determined. In our algorithm, the refactorings in this part is called the analyzed refactorings. The second part includes the refactorings which have not been analyzed. In this part, the positions of refactorings are unknown and need to analyze. The initial state of optimization, the number of the analyzed refactorings is zero; while the number of the unanalyzed refactorings is the number of rest of refactorings. We start to analyze the first refactoring of unanalyzed refactorings with the rest of the unanalyzed refactorings. We reorder the unanalyzed refactorings based on two situations:

- When we have analyzed the first refactoring of part of unanalyzed refactorings, we move it to the part of analyzed refactorings. Then we iterative this process until all refactorings is reordered.
- During the iterative process of the first situation, if we have found that one unanalyzed refactoring is depend on the first refactoring of part of unanalyzed refactoring, we move it to the part of reorder refactorings.

In summary, this algorithm shows the process of reordering the sequence by determining the updated position of the refactorings. The best order of the refactoring is to expose more optimization potentials. So far, our implementation is a simplified prototype by simplified dependency analysis. In practical application, if there are complex potential dependency detections, we should disallow reordering the location of relevant refactorings to guarantee that reordered sequence is a legal sequence.

The second algorithm is checking the pairs of refactorings step by step within the reordered sequence to find the chance of applying optimization rules. The basic idea is based on the two aspects:

- We should defined the optimization rules in the class *Compositionrules* such as $[RC1_{C1 \Rightarrow C2} \Rightarrow RC2_{C2 \Rightarrow C3} \equiv RCnew_{C1 \Rightarrow C3}]$, $[RM1_{C.M1 \Rightarrow C.M2} \Rightarrow RM2_{C.M2 \Rightarrow C.M3} \equiv RMnew_{C.M1 \Rightarrow C.M3}]$, $[RF1_{C.f1 \Rightarrow C.f2} \Rightarrow RF2_{C.f2 \Rightarrow C.f3} \equiv RFnew_{C.f1 \Rightarrow C.f3}]$, $[EI_{C1 \Rightarrow I1} \Rightarrow RC_{I1 \Rightarrow I2} \equiv EInew_{C1 \Rightarrow I2}]$ and $[RM_{C.M1 \Rightarrow C.M2} \Rightarrow IM_{C.M2 \Rightarrow Null} \equiv IMnew_{C.M1 \Rightarrow Null}]$, etc.

Algorithm 2 Optimizing the sequence of refactorings by applying rules

Input:

The reordered sequence of refactorings(SEQ': $< R_1, R_2, ...R_n >$) generated from Algorithm1.

Output:

The optimized sequence of refactorings(SEQ'') after applying the optimization rules.

```
1: Create an empty $SEQ''$;
2: for $i = 0$; $i < SEQ'.length$; $i++$ do
3:   if $i < SEQ'.length - 1$ and $R_i \circ R_{i+1}$ and $PostKey$ of $R_i$ = $PreKey$ of $R_{i+1}$ then
4:     Create new $R$ by merging the $R_i \circ R_{i+1}$;
5:     Set Arguments, $PreKey$ and $PostKey$, etc. to $R$;
6:     Add $R$ into $SEQ''$;
7:     i++;
8:   else
9:     Add $R_i$ into $SEQ''$;
10:  end if
11: end for
```

- When some pairs of refactorings can match the predefined optimization rules in the class *OptimizationRules*. We can generate the updated refactorings for simplifying the sequence. In other words, the number of refactorings are reduced so that the user-defined sequence can be optimized. The process of this algorithm start to create an new sequence, which is step by step filled the original refactorings or updated refactorings by processing refactorings of the reordered sequence. Then the refactorings of this sequence and the equation file are rewritten.

4.3 The Limitations of Application

So far, our prototype of *RFMoptimizer* can successfully be applied to a number of sequences of refactorings. But it is still not so robust to deal with all disadvantages sequences. The limitations of application are described as follows:

For one thing, our implementation only integrates five optimization rules. If optimization potentials of disadvantages sequences are based on these optimization rules, our prototype is available for optimizing these sequences. Otherwise, our prototype cannot perform the optimization.

For another, our implementation did not integrate the analysis of concrete program. Therefore, when we execute the step of reordering refactorings in the given sequence, there exist some situations cannot be optimized. In other words, the physical optimization is not implemented in our prototype.

Chapter 5

Case Studies

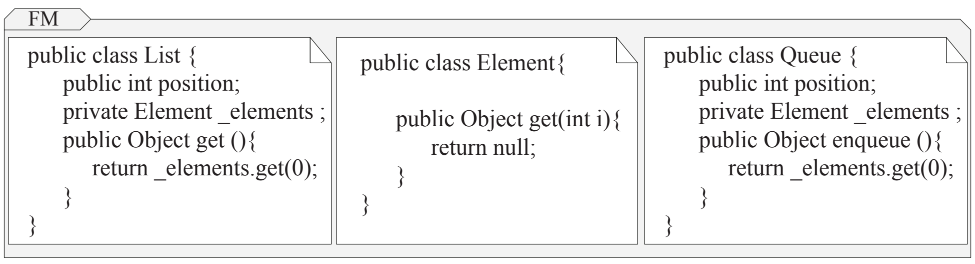
In the preceding chapter, we implemented a prototype of *RFMoptimizer* for optimization of RFMs. To verify the effect of optimization, we now report on four different types of case studies in this chapter. We will introduce them in detail in Section 5.1. Moreover, we apply *RFMoptimizer* to these motivating cases and evaluate corresponding the optimization effect and optimization process in Section 5.2.

5.1 Motivating Cases

In this section, we firstly introduce a simplified program "List"(19 lines of source code) which is rather small but enough to demonstrate our basic concept. The second case is derived from "TankWar" game, which is a software produce line demonstration developed by feature oriented design. Its size(1k lines of source code) is larger than the first case. The third case is eclipse library "workbench.texteditor" (17k lines of source code), which is already moved into a normal feature module in paper [KBK09a]. The final case is from the paper [KBA09] about the "ZipMe"(3k lines of source code) which is a library to access ZIP archives. Since *RFMoptimizer* focuses on optimizing sequences of RFMs, the source code of four case studies have already been derived to one normal feature module.

SimplifiedList.

The first case is simple, but it is the best one to demonstrate our concept. We start to show the base program in a normal feature module. To illustrate, Figure 5.1 depicts three classes, *List*, *Element* and *Queue* in the normal feature module.



```
public class List {
    public int position;
    private Element _elements ;
    public Object get (){
        return _elements.get(0);
    }
}
```

```
public class Element{

    public Object get(int i){
        return null;
    }
}
```

```
public class Queue {
    public int position;
    private Element _elements ;
    public Object enqueue (){
        return _elements.get(0);
    }
}
```

Figure 5.1: The base program in the feature module in Case **SimplifiedList**

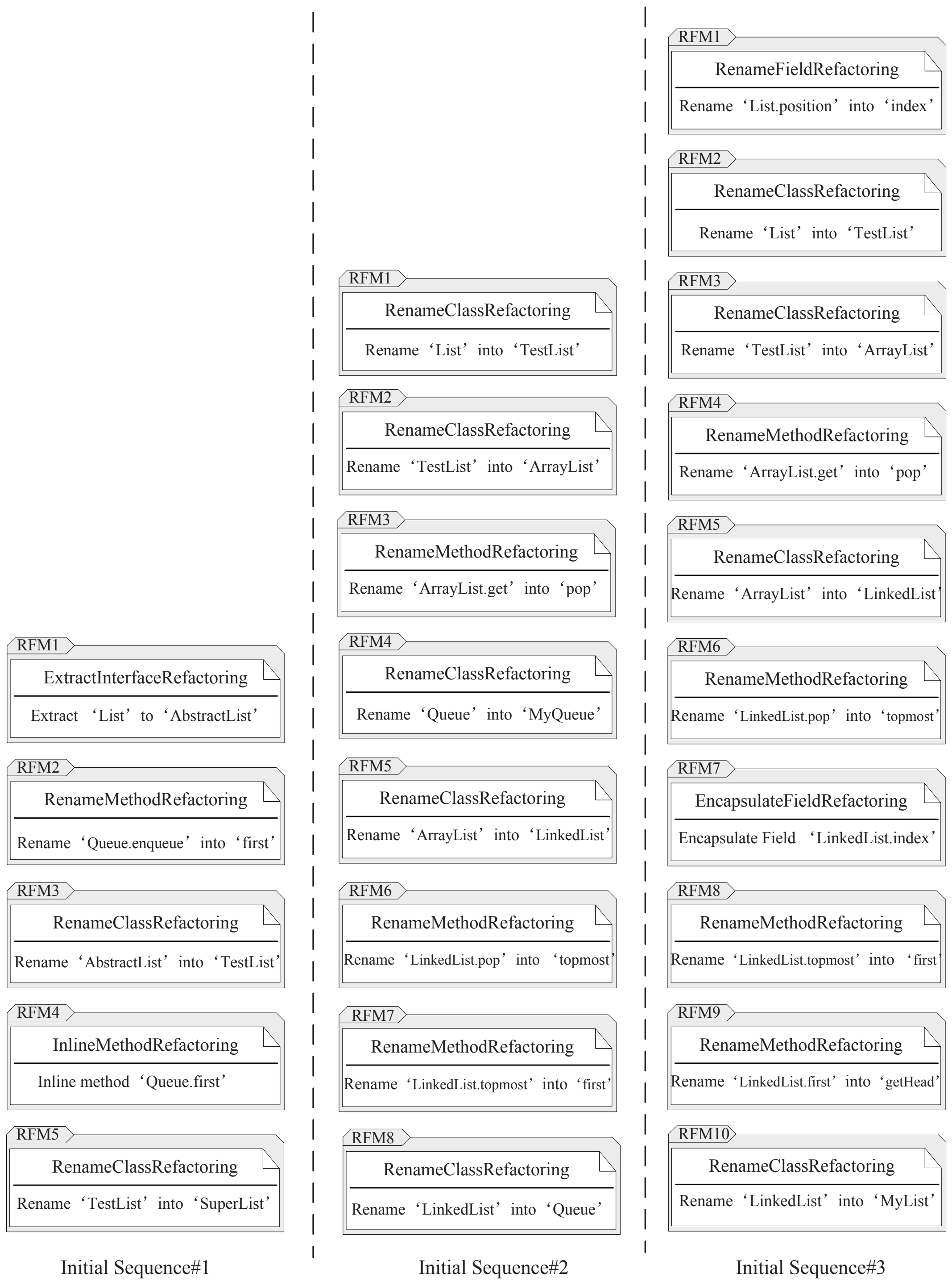

RFM1
ExtractInterfaceRefactoring
Extract ‘List’ to ‘AbstractList’
RFM2
RenameMethodRefactoring
Rename ‘Queue.enqueue’ into ‘first’
RFM3
RenameClassRefactoring
Rename ‘AbstractList’ into ‘TestList’
RFM4
InlineMethodRefactoring
Inline method ‘Queue.first’
RFM5
RenameClassRefactoring
Rename ‘TestList’ into ‘SuperList’
Initial Sequence#1
RFM1
RenameClassRefactoring
Rename ‘List’ into ‘TestList’
RFM2
RenameClassRefactoring
Rename ‘TestList’ into ‘ArrayList’
RFM3
RenameMethodRefactoring
Rename ‘ArrayList.get’ into ‘pop’
RFM4
RenameClassRefactoring
Rename ‘Queue’ into ‘MyQueue’
RFM5
RenameClassRefactoring
Rename ‘ArrayList’ into ‘LinkedList’
RFM6
RenameMethodRefactoring
Rename ‘LinkedList.pop’ into ‘topmost’
RFM7
RenameMethodRefactoring
Rename ‘LinkedList.topmost’ into ‘first’
RFM8
RenameClassRefactoring
Rename ‘LinkedList’ into ‘Queue’
Initial Sequence#2
RFM1
RenameFieldRefactoring
Rename ‘List.position’ into ‘index’
RFM2
RenameClassRefactoring
Rename ‘List’ into ‘TestList’
RFM3
RenameClassRefactoring
Rename ‘TestList’ into ‘ArrayList’
RFM4
RenameMethodRefactoring
Rename ‘ArrayList.get’ into ‘pop’
RFM5
RenameClassRefactoring
Rename ‘ArrayList’ into ‘LinkedList’
RFM6
RenameMethodRefactoring
Rename ‘LinkedList.pop’ into ‘topmost’
RFM7
EncapsulateFieldRefactoring
Encapsulate Field ‘LinkedList.index’
RFM8
RenameMethodRefactoring
Rename ‘LinkedList.topmost’ into ‘first’
RFM9
RenameMethodRefactoring
Rename ‘LinkedList.first’ into ‘getHead’
RFM10
RenameClassRefactoring
Rename ‘LinkedList’ into ‘MyList’
Initial Sequence#3

Figure 5.2: The three initial sequences of Case **SimplifiedList**

Then, we design three different sequences in Figure 5.2 and compose them into the normal feature module. Firstly, the sequence#1 consists of five specified refactorings: $EI_{List \Rightarrow AbstractList}$, $RM_{Queue.enqueue \Rightarrow Queue.first}$, $RC_{AbstractList \Rightarrow TestList}$, $IM_{Queue.first \Rightarrow ...}$ and $RC_{TestList \Rightarrow SuperList}$. It is designed to demonstrate that our optimization rules which can be successfully applied to different types of refactoring. For example, $EI_{List \Rightarrow AbstractList} \Rightarrow RC_{AbstractList \Rightarrow TestList} \equiv EI_{List \Rightarrow TestList}$ and $RM_{Queue.enqueue \Rightarrow Queue.first} \Rightarrow IM_{Queue.first \Rightarrow ...} \equiv IM_{Queue.enqueue \Rightarrow ...}$ are implemented optimization rules which can merge different types of refactorings. Optimizing the sequence#1 can prove that our optimization approach has an extension potential by integrating more optimization rules. Secondly, in sequence#2, there is a set-up-deletion-of-identifier dependency between $RC_{Queue \Rightarrow MyQueue}$ in RFM4 and $RC_{LinkedList \Rightarrow Queue}$ in RFM8. Based on the concept of set-up-negative-identifier dependency, before executed, $RC_{LinkedList \Rightarrow Queue}$ requires $RC_{Queue \Rightarrow MyQueue}$ to rename the class Queue. Our implementation is able to address this kind of dependency by disallowing reordering the $RC_{LinkedList \Rightarrow Queue}$. In the evaluation section, we will see the optimized sequence#2 as well as expose the optimization effect about this sequence. Lastly, the design of the sequence#3 is to evaluate the optimization effect by compared with the following case.

Before introducing the next case, We firstly start the *RFMcomposer*[1] to measure the compilation time for composing these initial sequences of RFMs. After running, these sequences of RFMs can be performed successfully onto the feature module. The average compilation time[2] of sequence#1 is 12018.6 ms, the average compilation time of sequence#2 is 12840.7 ms and the average compilation time of sequence#3 is 16359.3 ms.

TankWar.

The second case about **TankWar** is a feature oriented program for demonstrating concepts of software product lines. The base program in the normal feature module contains twelve classes. The scale of this base program is larger than the base program from the case **SimplifiedList**. Similar to the case **SimplifiedList**, we design ten RFMs that encapsulate the same type of refactorings as the case **SimplifiedList** and compose them as follows:

1. The field *TankManager.map* is renamed into *gameMap* in RFM1.
2. The class *Maler* is renamed into *Drawing* in RFM2.
3. The class *Drawing* is renamed into *DrawObjects* in RFM3.
4. The method *DrawObjects.getImage* is renamed into *storeImage* in RFM4.
5. The class *DrawObjects* is renamed into *DrawComponent* in RFM5.
6. The method *DrawComponent.storeImage* is renamed into *defineImage* in RFM6.
7. The field *TankManager.gameMap* is encapsulated in RFM7.

[1]*RFMcomposer* is an existing compose tool which can compose RFMs onto the normal feature module.

[2]The compilation environment is on Microsoft Window XP Home Edition sp2 and the hardware's configuration is Intel(R) Core(TM)2 CPU T550 @ 1.66GHz 980MHz, RAM 0.99 GB.

8. The method *DrawComponent.defineImage* is renamed into *defineImage1* in RFM8.
9. The method *DrawComponent.defineImage1* is renamed into *defineImage2* in RFM9.
10. The class *DrawComponent* is renamed into *DrawAll* in RFM10.

We also start the *RFMcomposer* to measure the compilation time for composing this original sequence of RFMs. The average compilation time of this sequence is 31934.2 ms. It is obvious that this value is larger than that of the first case. The reason is that the base program in this case is larger than that of the first case. We expect to get a better optimization effect for the larger program. To further illustrate, we continue to apply the same type of refactorings to a much larger program in the next case and evaluate the corresponding result.

workbench.texteditor.

The third case is a large scale program about '*workbench.texteditor*'. The base program in the normal feature module is derived from the eclipse library '*workbench.texteditor*'(16K lines of source code). In this case, we design three sequences of RFMs.

The sequence#1 is similar to the previous two cases. It consists of ten RFMs as follows:

1. The field *DefaultCellComputer.levenstein* is renamed into *levenshtein* in RFM1.
2. The class *Levenstein* is renamed into *Levenshtein* in RFM2.
3. The class *Levenshtein* is renamed into *Levenshteina* in RFM3.
4. The method *Levenshteina.createLevenstein* is renamed into *createLevenshtein* in RFM4.
5. The class *Levenshteina* is renamed into *Levenshteinb* in RFM5.
6. The method *Levenshteinb.createLevenshtein* is renamed into *createLevenshtein1* in RFM6.
7. The field *DefaultCellComputer.levenshtein* is encapsulated in RFM7.
8. The method *Levenshteinb.createLevenshtein1* is renamed into *createLevenshtein2* in RFM8.
9. The method *Levenshteinb.createLevenshtein2* is renamed into *createLevenshtein3* in RFM9.
10. The class *Levenshteinb* is renamed into *Levenshteinc* in RFM10.

We also start the *RFMcomposer* to measure the compilation time for composing this original sequence of RFMs. The average compilation time of this sequence is 172162.4 ms. It further proves the compilation time of a sequence is based on the scale of the program. We expect to reduce more compilation time by optimizing the unoptimized sequence. Besides, in concept, the compilation time of a sequence depends on the number of RFMs with optimization potential in this sequence. Therefore, we expose two extra sequences to demonstrate this assertion.

The sequence#2 is still composed onto this base program and consists of seventeen RFMs as follows:

1. The class *Levenstein* is renamed to *Levenshtein* in RFM1.
2. The field *DefaultCellComputer.levenstein* is renamed to *levenshtein* in RFM2.
3. The field *OptimizedCellComputer.levenstein* is renamed to *levenshtein* in RFM3.
4. The field *OptimizedCellComputer.levenstein* is renamed to *levenshteina* in RFM4.
5. The field *OptimizedCellComputer.levensteina* is renamed to *levenshteinb* in RFM5.
6. The field *OptimizedCellComputer.levensteinb* is renamed to *levenshteinc* in RFM6.
7. The field *OptimizedCellComputer.levensteinc* is renamed to *levenshteind* in RFM7.
8. The field *OptimizedCellComputer.levensteind* is renamed to *levenshteine* in RFM8.
9. The field *OptimizedCellComputer.levensteine* is renamed to *levenshteinf* in RFM9.
10. The field *OptimizedCellComputer.levensteinf* is renamed to *levenshteing* in RFM10.
11. The class *Levenshtein* is renamed to *Levenshteina* in RFM11.
12. The class *Levenshteina* is renamed to *Levenshteinb* in RFM12.
13. The class *Levenshteinb* is renamed to *Levenshteinc* in RFM13.
14. The class *Levenshteinc* is renamed to *Levenshteind* in RFM14.
15. The class *Levenshteind* is renamed to *Levenshteine* in RFM15.
16. The class *Levenshteine* is renamed to *Levenshteinf* in RFM16.
17. The class *Levenshteinf* is renamed to *Levenshteing* in RFM17.

We also start the *RFMcomposer* to measure the compilation time for composing this original sequence of RFMs. The average compilation time of this sequence is 253831.2 ms.

The sequence#3 has already been designed in paper [KBK09a] consists of fifty-five RFMs. The fifty-five RFMs are composed successfully onto the base feature module in the following order:

1. The class *Levenstein* is renamed to *Levenshtein* in RFM1.
2. The field *DefaultCellComputer.levenstein* is renamed to *levenshtein* in RFM2.

3. The field *OptimizedCellComputer.levenstein* is renamed to *levenshtein* in RFM3.
4. The field *OptimizedCellComputer.levenstein* is renamed to *levenshteina* in RFM4.
5. The field *OptimizedCellComputer.levensteina* is renamed to *levenshteinb* in RFM5.
6. ... 28.
29. The field *OptimizedCellComputer.levensteiny* is renamed to *levenshteinz* in RFM29.
30. The class *Levenshtein* is renamed to *Levenshteina* in RFM30.
31. The class *Levenshteina* is renamed to *Levenshteinb* in RFM31.
32. ... 54.
55. The class *Levenshteiny* is renamed to *Levenshteinz* in RFM55.

We also start the *RFMcomposer* to measure the compilation time for composing this original sequence of RFMs. The average compilation time of this sequence is 769617.5 ms.

ZipMe Library.

The final case about **ZipMe** is already derived into the normal feature module in paper[KBA09] with three RFMs. The three RFMs and their execution order are described as follows:

1. The class *ZipArchive* is renamed to *ZipFile* by performing a RFM.
2. The class *ZipFile* from the package *zipme* is moved to the package *jazzlib*
3. The class *ZipEntry* from the package *zipme* is moved to the package *jazzlib*

We also start the *RFMcomposer* to measure the compilation time for composing this original sequence of RFMs. This original sequence of RFMs can be performed successfully onto the feature module. The average compilation time of this sequence is 20461 ms.

After describing eight sequences of four cases and the compilation time of these original sequences, we will reveal the optimized sequences and evaluate the compilation time of optimizing and composing these sequences in the next section. At the same time, we also discuss some limitations in these cases and evaluate optimization effects.

5.2 Evaluation

By performing the *RFMoptimizer* to optimize the sequence of refactorings, we can produce new updated refactorings to replace some original refactorings. At the same time, the new compilation order of these refactorings is rewritten to the equation file of the existing compose tool. So far our prototype is independent of the *RFMcomposer*. The total optimization process of our prototype consists of loading RFMs, optimizing RFMs, rewriting updated RFMs and their corresponding order, as well as deleting temporary files. We will evaluate the compilation time of the four parts of optimization process and total optimization process for each case in this section. After performing optimization process, we start the *RFMcomposer* to compose the optimized sequence of new RFMs onto the feature module based on the new execution order of these new RFMs. Using this way, we can get the compilation time of composing new sequence. Calculating the optimization effect is the main intention of our evaluation, which is given by the following equation:

$$\text{Optimization Effect} = (1 - \frac{T_{optimization} + T_{newcomposition}}{T_{oldcomposition}}) * 100\% \tag{5.1}$$

In the equation 5.1, the optimization effect is the percentage by which the compilation time of composition can be reduced. $T_{oldcomposition}$ denotes the compilation time of composing the original sequence. $T_{newcomposition}$ denotes the compilation time of composing the optimized sequence. $T_{optimization}$ denotes the compilation time of optimization process.

For the case **SimplifiedList**, three optimized sequences consist of some merged refactorings and some original refactorings. These refactorings are encapsulated in the corresponding RFMs and the optimized composition order of RFMs, calculated by our prototype, shown in Figure 5.3 are top to down.

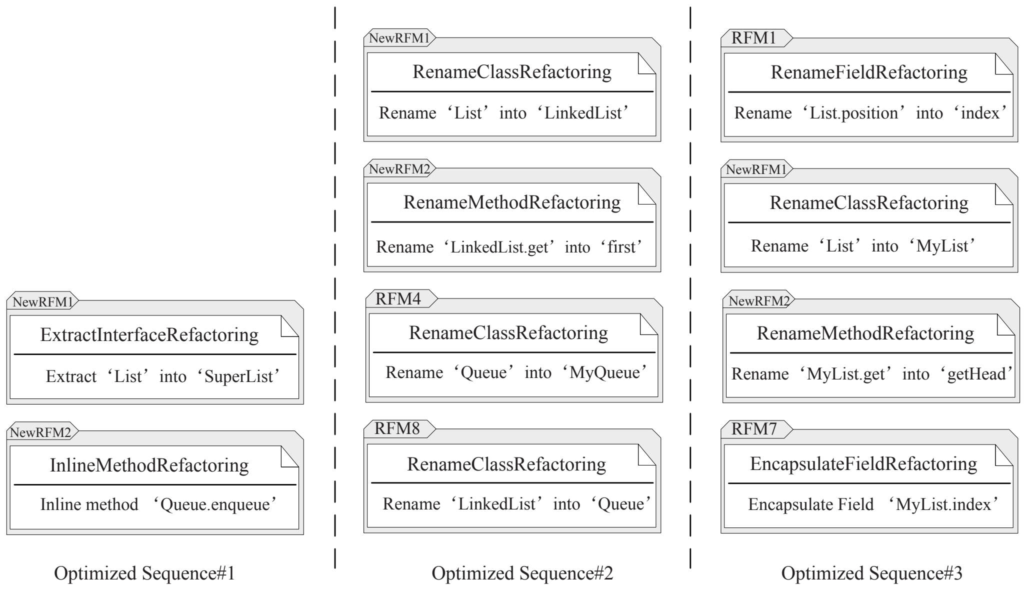

NewRFM1
ExtractInterfaceRefactoring
Extract 'List' into 'SuperList'
NewRFM2
InlineMethodRefactoring
Inline method 'Queue.enqueue'
Optimized Sequence#1
NewRFM1
RenameClassRefactoring
Rename 'List' into 'LinkedList'
NewRFM2
RenameMethodRefactoring
Rename 'LinkedList.get' into 'first'
RFM4
RenameClassRefactoring
Rename 'Queue' into 'MyQueue'
RFM8
RenameClassRefactoring
Rename 'LinkedList' into 'Queue'
Optimized Sequence#2
RFM1
RenameFieldRefactoring
Rename 'List.position' into 'index'
NewRFM1
RenameClassRefactoring
Rename 'List' into 'MyList'
NewRFM2
RenameMethodRefactoring
Rename 'MyList.get' into 'getHead'
RFM7
EncapsulateFieldRefactoring
Encapsulate Field 'MyList.index'
Optimized Sequence#3

Figure 5.3: The three optimized sequences of Case **SimplifiedList**

Because of the limitation of our prototype's implementation, our prototype now just optimize a sequence by matching existing predefined optimization rules. It could be extended to include more optimization rules but now unimplemented optimization rules are not measured. As illustrated in Figure 5.3, the pair of 'Rename Field' refactoring and 'Encapsulated Field' refactoring are not composed in optimized sequence#3. The other cases also have this limitation. However, the existing optimization rules can be applied to optimize the rest of refactorings for three sequences in this case.

The average compilation time of composing optimized sequence#1 is 9870.4 ms, the average compilation time of composing optimized sequence#2 is 9546.8 ms and the average compilation time of composing optimized sequence#3 is 10412.3 ms. these values which are less than that of unoptimized sequences. But we also consider the compilation time of optimization process itself. The average compilation time of total optimization process of sequence#1 is 8934.6 ms, that of sequence#2 is 9401.4 ms and sequence#3 is 9074.6 ms. Based on the equation 5.1, the optimization effect of sequence#1 = -56.4%, that of sequence#2 = -47.6% and that of sequence#3 = -19.1%. It shows negative optimization effects in optimization process. There are two factors lead to this kind of negative optimization effect. First is that we repeat computing the cost of loading RFMs during total optimization process. In fact, the process of loading RFMs in our prototype can be eliminated if our implementation will be integrated to the *RFMcomposer*. Second is that the cost of rewriting the optimized sequence of RFMs also could be ignored, when the *RFMoptimizer* will be integrated into the existing composition tool. In other words, the loaded RFMs of the optimized sequence can directly be executed without writing them to hard disk after *RFMoptimizer* integrated into the existing composition tool. If we remove the cost of loading RFMs and rewriting RFMs/equation file, the optimization effect of sequence#1 without loading RFMs and rewriting RFMs/equation file= 17.4%, that of sequence#2 = 25.1% and that of sequence#3 = 35.8%. Besides, the compilation time of composing a RFM is based on the scale of a base program. The larger the scale of the base program is, the more compilation time of composing a RFM it needs. If the scale of program is too small, the optimization effect could be not obvious. In the following cases, we discuss the positive optimization effect for two larger base programs in the next two cases.

For the case **TankWar**, the new generated sequence consists of two new refactorings generated by *RFMoptimizer* and two original refactorings. The four refactorings are encapsulated in the corresponding RFMs and the new composition order of RFMs are describe as follows:

1. The field *TankManager.map* is renamed into *gameMap* in RFM1.
2. The class *Maler* is renamed into *DrawAll* in NewRFM1.
3. The method *DrawAll.getImage* is renamed into *defineImage2* in NewRFM2.
4. The field *TankManager.gameMap* is encapsulated in RFM7.

Similar to the evaluation of the case “SimplifiedList”, the compilation time of composing optimized sequence is 14093.6 ms which is less than that of the unoptimized sequence. But we also consider the compilation time of optimization process itself. The average compilation time of total optimization process is 8206.3 ms and that of optimization process without loading RFMs and rewriting RFMs/equation file is 89 ms. Based on the equation 5.1, the optimization effect = 30.1%. The optimization effect without loading RFMs and rewriting RFMs/equation file = 55.6%.

If the scale of base program is larger than the base program of the case "SimplifiedList" and the composed RFMs are similar, the optimization effect is more obvious.

For the case **workbench.texteditor**, we have already designed three unoptimized sequences of RFMs for the base program derived from eclipse library "workbench.texteditor". After optimizing the original sequence#1, we generated the optimized sequence#1 that consists of two refactorings generated by *RFMoptimizer* and two original refactorings. The four refactorings are encapsulated in the corresponding RFMs and the new composition order of these RFMs is described as follows:

1. The field *DefaultCellComputer.levenstein* is renamed into *levenshtein* in RFM1.
2. The class *Levenstein* is renamed into *Levenshteinc* in NewRFM1.
3. The method *Levenshteinc.createLevenstein* is renamed into *createLevenshtein3* in NewRFM2.
4. The field *DefaultCellComputer.levenshtein* is encapsulated in RFM7.

Similar to the evaluation of the case "SimplifiedList" and the case "TankWar", the compilation time of composing optimized sequence#1 is 83561.1 ms which is much less than that of the unoptimized sequence. The average compilation time of total optimization process is 18749.9 ms and that of optimization process without loading RFMs and rewriting RFMs/equation file is 343.6 ms. Based on the equation 5.1, the optimization effect = 40.6%. The optimization effect without loading RFMs and rewriting RFMs/equation file= 51.3%. As we can see from these data, if types of refactorings encapsulated in the composed RFMs are similar, the optimization effect is based on the scale of base program. The larger the base program is, the better the optimization effect is. In the following two sequences in this case, we expect to see the relationship between the optimization effect and the number of RFMs.

After optimizing sequence#2, we generate optimized sequence#2 which consists of two new refactorings generated by *RFMoptimizer* and one original refactoring. The three refactorings are encapsulated in the corresponding RFMs and the new composition order of these RFMs is described as follows:

1. The 'Rename Class' refactoring encapsulated in NewRFM1 renames class *Levenstein* into *Levenshteing.*
2. The 'Rename Field' refactoring encapsulated in RFM2 renames field *DefaultCellComputer.levenstein* into *levenshtein.*
3. The 'Rename Field' refactoring encapsulated in NewRFM2 renames field *OptimizedCellComputer.levenstein* into *levenshteing.*

About the sequence#2, the compilation time of composing optimized sequence#2 is 59731.2 ms which is much less than that of the unoptimized sequence. The average compilation time of total optimization process is 18448.4 ms and that of optimization process without loading RFMs and rewriting RFMs/equation file is 241.9 ms. Based on the equation 5.1, the optimization effect = 69.2%. The optimization effect without loading RFMs and rewriting RFMs/equation file = 76.4%.

After optimizing sequence#3, we generate optimized sequence#3 which consists of two new refactorings generated by *RFMoptimizer* and one original refactoring. The three refactorings are encapsulated in the corresponding RFMs and the new composition order of these RFMs is described as follows:

1. The 'Rename Class' refactoring encapsulated in NewRFM1 renames class *Levenstein* into *Levenshteinz*.
2. The 'Rename Field' refactoring encapsulated in RFM2 renames field *DefaultCellComputer.levenstein* into *levenshtein*.
3. The 'Rename Field' refactoring encapsulated in NewRFM2 renames field *OptimizedCellComputer.levenstein* into *levenshteinz*.

About the sequence#3, the compilation time of composing optimized sequence#3 is 61292.1 ms which is much less than that of the unoptimized sequence. The average compilation time of total optimization process is 77632 ms and that of optimization process without loading RFMs and rewriting RFMs/equation file is 1698.3 ms. Based on the equation 5.1, the optimization effect = 81.9%. The optimization effect without loading RFMs = 91.8%. As we can see from these data, if the scale of the program is similar, the optimization effect is based on the number of RFMs which expose optimization potential. The more RFMs with optimization potential, the better the optimization effect is.

For the case **ZipMe Library**, we cannot produce optimized sequence, because the relationship among three refactorings do not expose the optimization chance. Therefore, the three refactorings and the composition order of RFMs are the same to the original sequence are described as follows:

1. The class *ZipArchive* is renamed to *ZipFile* by performing a RFM.
2. The class *ZipFile* from the package is moved *zipme* to the package *jazzlib*
3. The class *ZipEntry* from the package is moved *zipme* to the package *jazzlib*

About evaluation of this sequence, the compilation time of composing optimized sequence is 20281.4 ms which is approximately equal to 20461 ms of the unoptimized sequence. The average compilation time of total optimization process is 7867.1 ms which is checking the optimization potential in this sequence. Based on the equation 5.1, the optimization effect = -37.5%. The optimization effect without loading RFMs and rewriting RFMs/equation file = 0.34%. As we can see from data, there is not an optimization potential. After optimizing the sequence, the sequence is unchanged and the compilation time of composing the sequence is a constant value. The compilation time of optimization process causes a few negative effects.

In summary, when we compose the unoptimized and the optimized sequences of RFMs into the base program in the normal feature module for the four cases respectively, the results of composition is same. Furthermore, we can capture the corresponding compilation time of optimizing and composing sequences to observe the optimization effect. We summarize the relevant data[3] in Table 5.1 and Table 5.2. More detailed data can be referred in Appendix C. As we can see from Table 5.2 and Figure 5.4, the optimization effect is based on the scale of the base program and the number of RFMs which expose optimization potentials.

	Composition Time		Optimization Time	
8 sequences in 4 case studies	$T_{unoptimized}$	$T_{optimized}$	$T_{optimization}$	$T'_{optimization}$[4]
SimplifiedList#1$_{(5RFMs)}$	12018.6 ms	9870.4 ms	8934.6 ms	59.6 ms
SimplifiedList#2$_{(8RFMs)}$	12840.7 ms	9546.8 ms	9401.4 ms	73.7 ms
SimplifiedList#3$_{(10RFMs)}$	16359.3 ms	10412.3 ms	9074.6 ms	82.6 ms
TankWar$_{(10RFMs)}$	31934.2 ms	14093.6 ms	8206.3 ms	89 ms
workbench.texteditor#1$_{(10RFMs)}$	172162.4 ms	83561.1 ms	18749.9 ms	343.6 ms
workbench.texteditor#2$_{(17RFMs)}$	253831.2 ms	59731.2 ms	18448.4 ms	241.9 ms
workbench.texteditor#3$_{(55RFMs)}$	769617.5 ms	61292.1 ms	45103 ms	1698.3 ms
ZipMe Library$_{(10RFMs)}$	20461 ms	20281.4 ms	7867.1 ms	109.2 ms

Table 5.1: The unoptimized/optimized composition time and the optimization time

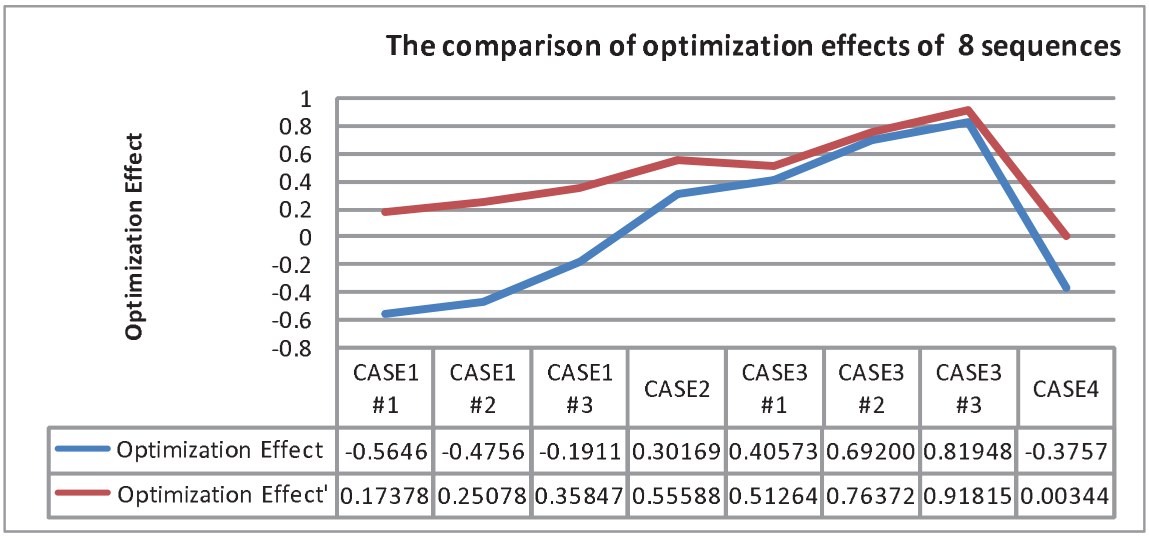
8 sequences in 4 case studies	Optimization Effect	Optimization Effect′[5]
SimplifiedList#1$_{(5RFMs)}$	-56.5%	17.4%
SimplifiedList#2$_{(8RFMs)}$	-47.6%	25.1%
SimplifiedList#3$_{(10RFMs)}$	-19.1%	35.8%
TankWar$_{(10RFMs)}$	30.2%	55.6%
workbench.texteditor#1$_{(10RFMs)}$	40.5%	51.3%
workbench.texteditor#2$_{(17RFMs)}$	69.2%	76.4%
workbench.texteditor#3$_{(55RFMs)}$	81.9%	91.8%
ZipMe Library$_{(10RFMs)}$	-37.6%	0.34%

Table 5.2: The comparison of the optimization effect for given sequences

[3]The evaluation environment is on Microsoft Window XP Home Edition sp2 and the hardware's configuration is Intel(R) Core(TM)2 CPU T550 @ 1.66GHz 980MHz, RAM 0.99 GB.

[4]$T_{optimization}$ denotes the compilation time of total optimization process and $T'_{optimization}$ denotes the compilation time of optimization process without loading RFMs and rewriting RFMs/equantion file.

[5]Optimization Effect denotes the compilation time of total optimization process and Optimization Effect′ denotes the compilation time of optimization process without loading RFMs and rewriting RFM-s/equantion file.



	CASE1 #1	CASE1 #2	CASE1 #3	CASE2	CASE3 #1	CASE3 #2	CASE3 #3	CASE4
🟦 Optimization Effect	-0.5646	-0.4756	-0.1911	0.30169	0.40573	0.69200	0.81948	-0.3757
🟥 Optimization Effect'	0.17378	0.25078	0.35847	0.55588	0.51264	0.76372	0.91815	0.00344

Figure 5.4: The comparison of the optimization effect for given sequences

Chapter 6

Related Work

The concepts presented in this thesis related to work from several fields: program transformation, program synthesis, artificial intelligence (AI) planning and more. In this chapter, we give a brief overview over category theory of computational design, relational query optimization of database, static composition of refactorings, detection of the relationships among refactorings, AI planning for optimization of refactorings and an algebra of patches.

Category theory for computational design.

Category Theory is a general theory of mathematical structures and their relationships[Pie91][Ste97]. Datory et al. use the concept of category theory as informal modeling language to explain Computational Design which is a paradigm where both program design and program synthesis are computations[BAS08]. Its foundation is that programs are values, transformations map programs to programs and operators map transformations to transformations[BAS08].

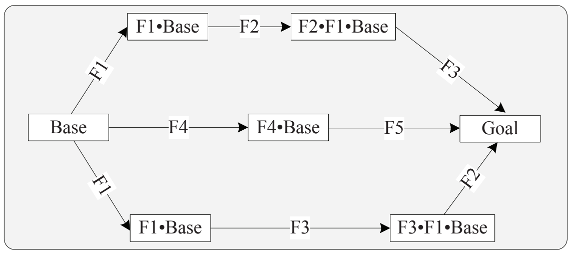
The paper[BAS08] describes how Software Product Lines(SPL) ideas map to categorical concepts. In the concept of Category Theory, a category is a collection of objects and arrows. As recursion is fundamental to category theory, an object is a domain of points and a single point can be seen as a special object. An arrow is a mapping relationship, a transformation is an implementation of an arrow as expressed by composition of features. From another perspective, The SPL is a set of similar programs. Programs are generated by composing different features, which are increments in program functionality[BAS08]. The transformation of adding features to the base program usually add codes (classes, methods, field and statements), while the transformation of refactorings to the base program usually create and delete code elements[KBK09a]. So Category Theory is also appropriate to support us for theoretical foundation of combination of refactorings. We extend the concept of refactorings to FOP and Category Theory terminology in Table 6.1.

Table 6.1 shows the terminological correspondence with concept of the refactorings. A program can be seen as an object in category theory and identifiers of program elements in the program can be seen as points in that category. Refactorings can be seen as arrows that map points (identifiers) of one object (transformed program) to points (identifiers) of another object.

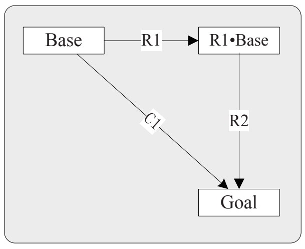
Paradigm	Point(Object)	Object	Arrow
FOP	program element	program	feature
RFMs	program element	program	refactoring in RFM

Table 6.1: Category Theory, FOP and RFMs Terminology

Based on Category Theory, the paper proposes the optimization of synthesizing programs[Bat08]. To illustrate in Figure 6.1(a), we can get the final program 'Goal' by composing a series of features to the program 'Base'. Goal=$F3 \bullet F2 \bullet F1 \bullet$Base or Goal=$F2 \bullet F3 \bullet F1 \bullet$Base or Goal=$F4 \bullet F5 \bullet$Base. We assume the $F2$ and $F1$ are commutative. At the same time, $F4$ and $F5$ can instead of $F1$, $F2$ and $F3$. So the concept of optimization problem of composition arrows is to determine how to find out the shortest composition paths[BAS08]. The RFMs that encapsulated refactorings also can be seen as normal features. Therefore these concepts are related to the optimizing sequences of RFMs. Similarly, to illustrate in Figure 6.1(b), we find out the shortest path 'C1' from composition paths of RFMs. In our approach, the basic concept of merging 'R1' and 'R2' to 'C1' is a foundation of optimization of sequences of RFMs.



(a) paths of composition normal features



(b) paths of composition RFMs

Figure 6.1: Different paths of composition features from 'Base' to 'Goal'

Relational query optimization in database

The principle of the optimization technique is from the Relational Query Optimization (RQO) that is a typical example of logical optimization and physical optimization[RGRG02].

In the logical level of RQO, the key insight is to transform query plan to relational algebra expressions. At the same time, several relational algebra equivalences can be applied to the relation algebra expression of query plan. For example, the two important equivalences that involves the selection operation, namely cascading of selections: $\sigma_{c1 \wedge c2 \wedge \ldots cn} \equiv \sigma_{c1}(\sigma_{c2}(\ldots \sigma_{cn}(R)))$ and commutative: $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$[RGRG02]. We also review one of the typical equivalences involve two or more operators, which is $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$[RGRG02]. We can commute a selection with a projection if the selection operation involves only attributes that are retained by the projection. Every attribute mentioned in the selection condition c must be included in the set of attribute a. In order to find alternative query plans, we can apply these predefined equivalences to relevant query operations of query plan in logical level. Relational algebra equivalences play a central role in identifying alternative plans to the original query plan. Distinguished with logical optimization level in database, there is no set theory supply deriving equivalences of our approach. But we can define the optimization rules by manually analyzing among standard refactorings. These optimization rules also can be seen as equivalences of pair of refactorings and used to derive the optimized sequence of refactorings to an original sequence.

If the estimated cost of the alternative plan is lower, we think it to be better than the original query plan. To estimate the cost of a plan, statistics of the relation instances in physical level should be considered. In physical level, there are different kinds of storage and access way about the data stored in a database. Take one typical example about access way, we can use direct scan a relation to find the relevant tuples or use an index to access only the relevant parts. Second, the algebraic operators used in the logical plan may have different alternative implementations. For example, the join operator that has many different implementations: nested loop join, hash join, sort merge join. Similar to our discussion about physical optimization, if the type of optimization rules is composition rule, it is impossible to optimize all refactorings which expose the optimization potential. At this time, the concrete program should be considered. Besides, due to the fact that our standard refactorings operate program elements of source code, the expression of source code also influence the efficiency of accessing program elements.

Static composition of refactorings.

Roberts' dissertation mentions composition of refactorings. He shows how to derive the precondition of a composite refactoring from the preconditions of the individual refactorings in a chain of refactorings[Rob99]. In our optimization approach, the optimization rules can be applied to pairs of refactorings that expose optimization potentials. Our optimization rules are divided to two categories: composition rules or simplification rules. Similar to the Roberts' concept, the composition rules can eliminate the precondition of a pair of refactorings and keep the two transformations. Distinguished from the Roberts' concept, the simplification rules not only eliminate the precondition of a pair of refactoring but also compose both transformations to one update transformation.

For a practical composition tool of primitive transformations, Guenter Kniesel et al come up with the concept of static composition of refactorings. The concept provides a formal model for automatic, program-independent composition of conditional program transformations[KK04]. In his concept, the conditional transformations can be composed from a limited set of primitive operations. Refactorings are a special form of conditional transformations whose preconditions ensure that the transformations are behavior preserving. They define a conditional transformation (CT) is a sequence of program transformations with a precondition. The precondition should be satisfied for their correct execution. If the precondition is satisfied, the transformation sequence is executed on a given program. The precondition may contain atomic conditions or complex ones created by conjunction, disjunction and negation of subconditions. Atomic conditions refer to the typical structure of a Java program (existence of classes, interfaces, fields, methods, parameters, etc.). Transformations can change such structures (add/delete/rename class, method, field, parameter, etc). Every transformation is associated with a transformation description which describes the effect of a transformation on the conditions that hold on a program. Furthermore, they automatically compute the joint preconditions for a chain of two transformations by two ways:

- Applying the transformation description of the first transformation to the precondition of the second one.
- Adding the derived precondition as an additional conjunction to the precondition of the first transformation.

If a chain contains more than two conditional transformations, then the process starts with the last precondition and the description of the preceding transformation and the process is iterated until it reaches the first conditional transformations in the chain.

This work describes how to derive a composite refactoring from primitive transformations in detail. The condition of primitive transformation is eliminated. But the transformation of the composite refactoring is conjunction of primitive transformation. Therefore, there are two different points from our work: one is that our work is to deal with the composition of standard refactorings and not to consider composition of primitive transformation. Another is that our work not only can merge the preconditions of refactorings but also can replace transformations of refactorings by one updated transformation of a standard refactoring.

Detection of the relationships among refactorings

At present, the related work about automatic analysis of dependencies of refactorings based on three types of program's representation namely abstract tree-based representation[Rob99], logic-based representation[Kni04] and graph-based representation[MVEDJ05]. So far there are some available tools to express and analyze program refactorings but no existing tool automatically to produce an optimized sequence of refactorings. In terms of our optimization approach, one key challenge is required to be considered. The challenge is to detect dependency and commutative relationships among standard refactorings. So far, we integrate one simplified way of exposing dependencies between two refactorings into our implementation of prototype. For instance, we detected a kind of dependencies for two refactorings by the $PostKey$ of the former refactoring is identical to the $PreKey$ of the latter refactoring. In fact, sometimes, detection of dependencies based on source code should be considered. For example, information of the overridden method could be acquired from the source code. Unfortunately, the textual representation of source code is difficult to express many of the relationships between program elements. After we investigated relevant researches, abstract syntax trees are suitable for expressing general Java program. At the same time, application of graph based expression and logic based expression are also created for representation of Java program to analyze the program elements.

Although the existing tools are still unavailable directly to do sequential analysis of program transformation, critical pair analysis and sequential dependency analysis on graph transformation are active research area [MVEDJ05]. For example, AGG system[1] that is a general purpose graph transformation tool developed by TU Berlin provides a possible solution for critical pair analysis and sequential dependency analysis. However it is still tedious procedures that we change object oriented semantic source program to graph-based expressions[MKR06a]. Moreover, critical pair analysis and sequential dependency analysis in the graph-based transformation need much execution time[MKR06a].

Another state of the art approach of critical pair analysis for refactorings is based on logic-based representation of Java program[2]. In this approach, the source program is translated to logic expressions. By Prolog language, which is a logic programming, checking the precondition and producing an updated program transformation are more efficient than graph-based transformation[MKR06a].

These work focus on dependency analysis for refactorings which is an important part of our approach. In our optimization approach, before applying optimization rules, we need to reorder refactorings to expose optimization potentials by analyzing the dependencies among these refactorings.

AI Planning for optimization of refactorings

During the reordering refactorings step of the optimization process, we want to find a reordered sequence of refactorings which exposes optimization potentials. This procedure is similar to rearranging individual actions from the starting state of world to the ending state in the field of artificial intelligent planning. There are some research works concentrating on applying the AI planning for performing the refactorings[P0́8]. Traditionally, automated planning is an artificial intelligence technique to generate sequences of actions that will achieve a certain goal when they are performed. Among all the existing planning approaches, Javier Perez introduces a proposal for generating refactoring plans with hierarchical task network (HTN) planning.

A refactoring plan is a specification of a sequence of refactorings which matches a program redesign proposal. It can be automatically executed to modify the program in order to obtain that desirable system redesign without changing the program's behavior.

A typical automated planning approach represents the current state of the world as a set of logical terms which are changed by application of operators. The operator also consists of a precondition and two separate sets of actions. A precondition specifies the condition under which they can be executed and actions specify how the operator modifies the state of the world. The goal is a list of terms which represents a certain state of the world we want to achieve. The task of the automated planning is to produce a plan as a sequence of operator instances that changes the world to achieve the desired goal in the final state. Furthermore, the actions of HTN planning composed by simple operators or by other tasks. Task networks allow including domain knowledge describing which subtasks should be performed to accomplish another one. HTN planning and forward search allows very expressive domain definitions which lead to detailed domains with a lot of

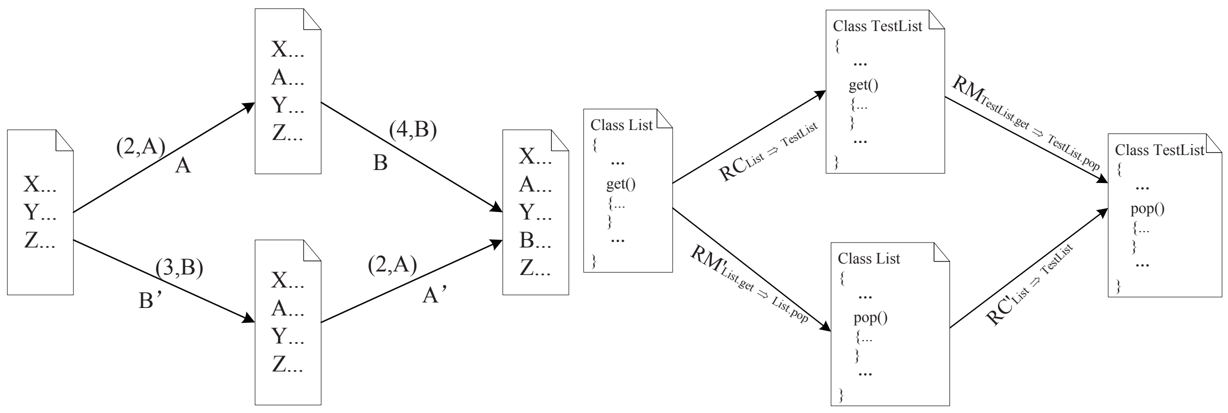
[1]http://user.cs.tu-berlin.de/ gragra/agg/
[2]http://roots.iai.uni-bonn.de/research/condor/

domain knowledge which can guide the planning process in a very efficient way. For finding out a better refactoring plan, the HTN planning should be tailored to search for plans which achieve a certain design structure. In the future, the feasibility of this concept might be implemented.

This kind of work focuses on finding out a refactoring plan by some AI algorithm. Its limitation is requiring a complete set of refactorings. It is impossible that generating a new refactoring instead of a group of refactorings which exposes optimization potential. But during the step of reordering refactorings of the optimization process, we do not require a new refactoring and only require a legal sequence of refactorings which exposes optimization potentials. The AI planning for refactorings is a complement to this step.

An algebra of patches

Darcs is a symmetric, distributed revision control system and working with sequences of patches. Lynagh introduces algebra of patches in his paper[Lyn06] to discuss how Darcs works, especially how to deal with conflicting patches. The core concept is to revert and commute patches for merging patches. The resulting sequence could shrink by merging sequences of patches. Similarly, our concept of commuting refactorings is to expose the optimization potential among some refactorings and merging these refactorings to a updated refactoring. Here, we compare the concept of commutation for patches with our concept as follows:


X... Y... Z...
(2,A)
A
X... A... Y... Z...
(4,B)
B
(3,B)
B'
X... A... Y... Z...
(2,A)
A'
X... A... Y... B... Z...
Class List { ... get() {... } ... }
$RC_{List \Rightarrow TestList}$
Class TestList { ... get() {... } ... }
$RM_{TestList.get \Rightarrow TestList.pop}$
$RM'_{List.get \Rightarrow List.pop}$
Class List { ... pop() {... } ... }
$RC'_{List \Rightarrow TestList}$
Class TestList { ... pop() {... } ... }

(a) The sequence of two patches[Lyn06] (b) The sequence of two refactorings

Figure 6.2: The example of commutation operation

The concept of commutation operator for the primitive patches is similar to our conditional commutation operator for the standard refactorings. The concrete example of commutation from the paper[Lyn06] is shown in Figure 6.2(a). Similarly, we describe our conditional commutation operator in Figure 6.2(b).

In Figure 6.2(a), the starting point is a repository containing the set of patches S which result in a file containing lines “X”, “Y” and “Z”. The effect of patch A is to add a line “A” as line 2 and the effect of patch B is to add a line “B” as line 4. Two new patches, B’ and A’, are created. The effect of patch B’ is to add a line “B” as line 3 and the effect of patch A is to add a line “A” as line 3. The patch sequences AB and B’A’ have the same start and the end repository state, but go through a different intermediate result. In other words, if we want to commute the A and the B, the corresponding actions require being adjusted. Similarly, in Figure 6.2(b), the starting point is a base program containing a class *List* and method *get*. The first refactoring RC is “rename class” refactoring which renames the class name “List” to “TestList”, the second refactoring RM is “rename method” refactoring which renames the method name “TestList.get” to “TestList.pop”. If we want to commute the two refactorings, updating corresponding parameters are necessary. The parameters of RM’ are changed from “TestList.get” and “TestList.pop” to “List.get” and “List.pop”. The parameters of RC’ are identical to that of RC. As we can see from Figure 6.2, the concept of our commutation operation is similar to the concept of commuting the patches.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The concept of RFMs is to combine refactorings with feature modules by defining refactorings in refactoring units that become elements of feature modules. Similar to the feature module sequences are composed to base programs, the refactorings can pre-defined in the RFMs and the RFMs also can be composed to base programs in a user-defined selection order to given base programs. However the user-defined original sequence of RFMs is composed by the existing compose tool might cause a high compilation effort because the existing compose tool write intermediate results into the disk during composing RFMs. In this thesis, we concentrated on providing the approach to optimize the sequences of refactorings so that the existing tool can eliminate the compilation effort when user-defined sequences of RFMs are disadvantageous sequences

In order to optimize sequences of refactorings, we presented a theoretical approach for optimization of sequences of refactorings from the RFMs. Firstly, we explored the standard refactorings, which could be all implemented in the RFMs. Based on the properties of legal sequences of RFMs, we revealed the potential relationships among the refactorings in given the sequence of refactorings from the corresponding RFMs. Besides, we defined relevant notions for refactorings and sequences of refactorings from a formal perspective. Furthermore, we revealed optimization rules for the pairs of refactorings among the standard refactorings. Finally, we presented the concept of logical optimization and physical optimization for sequences of refactorings.

As a proof of concept, we sketched an optimization prototype of optimizing sequences of refactorings. Based on implementation of this prototype, we design four case studies to evaluate our optimization approach and optimization effect. By evaluating, we showed that the prototype is available to optimize sequences of refactorings in four case studies. If the type of a sequence is the disadvantageous sequence, the compilation effort can be obviously eliminated by composing the optimized sequence of RFMs in these case studies. In addition, we reported some related works which inspire our approach.

The contribution of this thesis was significant to manually defined the optimization rules between standard refactorings. After optimization potentials are exposed by reordering a sequence of refactorings, these optimization rules can applied to optimize sequences of refactorings in logical level. However, there are still improvements for optimization in future work.

7.2 Future Work

So far, the optimization approach has already been integrated into the prototype of optimization tool (*RFMoptimizer*). We have already optimized some different sequences of refactorings in four case studies by using *RFMoptimizer*. Due to optimization effects in these cases is obvious, the concept of optimization approach is proved to be available for optimizing sequences of refactorings. But the *RFMoptimizer* cannot optimize all kinds of sequences of refactorings in practical application. Therefore, a central topic in future work is to improve the implementation of the prototype of optimization tool. The considered improvements related to the following four aspects:

1. Our concept is to optimize sequences of refactorings. These refactorings should be derived from all the standard refactorings[1]. But our implementation only integrated some parts of standard refactorings. In future work, we could integrate more standard refactorings, especially implement more optimization rules for different types of standard refactorings in practical application.

2. In our present implementation of prototype, we detected optimization potentials between the pair of refactorings by the $PostKey$ of the former refactoring is identical to the $PreKey$ of the latter refactoring. Essentially, this way of detection is one simplified way of exposing dependencies between two refactorings. In fact, the more complicate approaches of detecting dependencies among different refactorings need to be considered in the future. Moreover some kinds of dependencies even need to consider concrete programs. Involving the concrete programs could add the complexity and cost of detecting the dependencies. The concrete programs should find a suitable expression to improve efficiency of dependency analysis. For example, in the related work about dependency analysis among refactorings[MKR06b], the source code of the given program is transformed to graphical expressions or logical expressions. At the same time, it proves that the dependency analysis of refactorings based on logical expressions of concrete programs is more efficient than that based on other expressions of concrete program[MKR06b]. In future work, if considering the concrete programs, we could transform program elements to elements of logical knowledge base. This way could be more efficient than the way of accessing program element in the representation of other forms about source code.

3. We still need to predefine optimization rules for implementing merging refactorings of our prototype. In future work, we expect that automatic composition of the refactorings could be implemented in two aspects: For one thing, the concept of applying optimization rules not only cover the pair of the refactorings but also cover more than two refactorings each time. For another, the composition rules could be automatically implemented. The idea inspired by the paper about the static composition of refactorings[KK04]. Although his concept is about the primitive conditional transformation for user-defined standard refactorings, it is similar to composing individual refactorings to a large refactoring. By using this ways, we will not focus on predefine the simplification rules for the chain of refactorings.

[1]The standard refactorings are from the book[Fow99]

4. In the step of reordering refactorings of the optimization process, we want to find a reordered sequence of refactorings which exposes optimization potentials. This procedure is similar to rearranging individual actions from the starting state of world to the ending state in the field of artificial intelligent planning. A typical AI planning approach represents the current state of the world as a set of logical terms which are changed by application of operators[Ṕ08]. The operator also consists of a precondition and two separate sets of actions. Similar to the application of AI Planning in other fields, if starting state of program corresponds to the start of the task of AI planning, the final state of program transformation to the goal of the task and the refactorings correspond to the operators, we could integrate AI Planning into the optimization of sequences of refactorings. For instance, the related work proposed by Javier Perez, the HTN planning should be tailored to search for plans which achieve a certain design structure in order to find out a better refactoring plan. In future work, the way of finding out the reordered sequence of refactorings could be considered to integrate the AI planning.

Bibliography

[AB06] Apel, S.; Batory, D.: When to use features and aspects?: a case study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, S. 59–68. ACM, New York, NY, USA, 2006.

[BAS08] Batory, D.; Azanza, M.; Saraiva, a., J.: The objects and arrows of computational design. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, S. 1–20. Springer-Verlag, Berlin, Heidelberg, 2008.

[Bat05] Batory, D.: Feature models, grammars, and propositional formulas. In *to be published at Sofware Product Line Conference (SPLC 2005)*, 2005.

[Bat06] Batory, D.: A tutorial on feature oriented programming and the ahead tool suite. In *Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science, Band 4143, S. 3–35. Springer, 2006.

[Bat08] Batory, D.: Using modern mathematics as an fosd modeling language. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, S. 35–44. ACM, New York, NY, USA, 2008.

[Bec99] Beck, K.: *Extreme Programming Explained: Embrace Change.* Addison Wesley, 1999.

[BF01] Beck, K.; Fowler, M.: *Planning Extreme Programming.* Addison-Wesley, 2001.

[BSR03] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, S. 187–197. IEEE Computer Society, Washington, DC, USA, 2003.

[CE00] Czarnecki, K.; Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional, June 2000.

[CN00] Cinnéide, M. O.; Nixon, P.: Composite refactorings for java programs. In *Workshop on Formal Techniques for Java Programs*, S. 129–135, 2000.

[Coc06] Cockburn, A.: *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series).* Addison-Wesley Professional, 2006.

[Fow99] Fowler, M.: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[KBA08] Kuhlemann, M.; Batory, D.; Apel, S.: Refactoring feature modules. Technischer Bericht Nr. 15, Faculty of Computer Science, University of Magdeburg, 2008.

[KBA09] Kuhlemann, M.; Batory, D.; Apel, S.: Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse*, S. 106–115, 2009.

[KBK09a] Kuhlemann, M.; Batory, D.; Kästner, C.: Safe composition of non-monotonic features. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, 2009.

[KBK09b] Kuhlemann, M.; Batory, D.; Kästner, C.: Safe composition of refactoring feature modules. Technischer Bericht Nr. 7, Faculty of Computer Science, University of Magdeburg, 2009.

[KK04] Kniesel, G.; Koch, H.: Static composition of refactorings. *Science of Computer Programming*, Band 52, Nr. 1-3, S. 9 – 51, 2004.

[Kni04] Kniesel, G.: A logic foundation for program transformations. Technischer Bericht, CS Dept. III, University of Bonn, Germany, 2004.

[Kuh07] Kuhlemann, M.: Design Patterns Revisited. Technischer Bericht Nr. 2, Faculty of Computer Science, University of Magdeburg, 2007.

[Lyn06] Lynagh, I.: An algebra of patches. Technischer Bericht, 2006.

[MKR06a] Mens, T.; Kniesel, G.; Runge, O.: Transformation dependency analysis - a comparison of two approaches. *Série L'objet - logiciel, base de données, réseaux*, 2006.

[MKR06b] Mens, T.; Kniesel, G.; Runge, O.: Transformation dependency analysis - a comparison of two approaches. *Série L'objet - logiciel, base de données, réseaux*, 2006.

[MTR06] Mens, T.; Taentzer, G.; Runge, O.: Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 2006.

[MVEDJ05] Mens, T.; Van Eetvelde, N.; Demeyer, S.; Janssens, D.: Formalizing refactorings with graph transformations: Research articles. *J. Softw. Maint. Evol.*, Band 17, Nr. 4, S. 247–276, 2005.

[Opd92] Opdyke, W. F.: *Refactoring object-oriented frameworks.* Dissertation, Champaign, IL, USA, 1992.

[Ṕ08] Pérez, J.: Enabling refactoring with htn planning to improve the design smells correction activity. In *BENEVOL 2008 : The 7th BElgian-NEtherlands software eVOLution workshop*, 2008.

[Pie91] Pierce, B. C.: *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[RGRG02] Ramakrishnan, R.; Gehrke, J.; Ramakrishnan, R.; Gehrke, J.: *Database Management Systems*. McGraw-Hill Science/Engineering/Math, August 2002.

[Rob99] Roberts, D. B.: Practical analysis for refactoring. Technischer Bericht, Champaign, IL, USA, 1999.

[SKAP10] Siegmund, N.; Kuhlemann, M.; Apel, S.; Pukall, M.: Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, S. 115–122, January 2010.

[Ste97] Stephen: *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 1997.

Appendix A

Table A.1: 70 Basic Refactoring Operators [1]

RefOps	Transformation Description
RF	The “RenameField” refactoring transforms a name of the field to a new one.
RC	The “RenameClass” refactoring transforms a name of the class to a new one.
EM	The “ExtractMethod” refactoring turns the fragment of code into a method whose name is meaningful.
IM	The “InlineMethod” refactoring puts the method’s body into the body of its callers and remove the method.
IT	The “InlineTemp” refactoring replaces all references to that temp field with the expression.
RTwQ	The “ReplaceTempwithQuery” refactoring extracts the expression into a method. Replace all references to the temp with the new method.
IEV	The “IntroduceExplainingVariable” refactoring puts the result of the complex expression, or parts of the expression, in a new field.
STV	The “SplitTemporaryVariable” refactoring makes a separate temporary field for each assignment.
RAtP	The “RemoveAssignmentstoParameters” refactoring makes a temporary field instead of one parameter in some method.
RMwMO	The “ReplaceMethodwithMethodObject” refactoring turns the method into its own object so that all the local variables become fields on that object.
SA	The “SubstituteAlgorithm” refactoring replaces the body of the method with the new algorithm.
MM	The “MoveMethod” refactoring moves one method from the source class to the target class.
MF	The “MoveField” refactoring moves the field into the target class from the source class can change all its users.
ExtractC	The “ExtractClass” refactoring creates a new class and moves the relevant fields and methods from the old class into the new class.
IC	The “InlineClass” refactoring moves all its features into another class and deletes it.

next page. . .

RefOps	Transformation Description
HD	The “HideDelegate” refactoring creates methods on the server to hide the delegate.
RMM	The “RemoveMiddleMan” refactoring removes middle methods and get the client to call the delegate directly.
IFM	The “IntroduceForeignMethod” refactoring creates a method in the client class with an instance of the server class as its first argument.
ILE	The “IntroduceLocalExtension” refactoring creates a new class that contains these extra methods.
SEF	The “SelfEncapsulateField” refactoring creates getting and setting methods for the field and only use the two methods to access the field.
RDVwO	The “ReplaceDataValuewithObject” refactoring turns the field of data item into an object.
CVtR	The “ChangeValuetoReference” refactoring turns the value object into a reference object.
CRtV	The “ChangeReferencetoValue” refactoring turns the reference object into a value object.
RAwO	The “ReplaceArraywithObject” refactoring replaces the array with an object that has a field for each element.
DOD	The “DuplicateObservedData” refactoring copys the data to a new domain object and establishes an observer to synchronize the two pieces of data.
CUAB	The “ChangeUnidirectionalAssociationtoBidirectional” refactoring adds a field for back pointer and change modifiers to update both sets for new association between referring class and referred class.
CBAU	The “ChangeBidirectionalAssociationtoUnidirectional” refactoring drops the unneeded end of the association between referring class and referred class.
RMNwSC	The “ReplaceMagicNumberwithSymbolicConstant” refactoring creates a constant, name it after the meaning, and replace the number with it.
EF	“EncapsulateField” refactoring deals with the public field and make it private and provide accessors for other classes.
EC	The “EncapsulateCollection” refactoring makes it return a read-only view and provide add/remove methods replaces set method
RRwDC	The “ReplaceRecordwithDataClass” refactoring makes a dumb data object for the record.
RTCwC	The “ReplaceTypeCodewithClass” refactoring replaces the numeric type code by adding a new class.
RTCwS	The “ReplaceTypeCodewithSubclasses” refactoring replaces the immutable type code with new subclasses.
RTCwS/S	The “ReplaceTypeCodewithState/Strategy” refactoring replaces the type code with a state object.
RSwF	The “ReplaceSubclasswithFields” refactoring changes the methods to superclass fields and eliminates the subclasses.
DC	The “DecomposeConditional” refactoring extracts methods from the condition, then part, and else parts.

next page. . .

RefOps	Transformation Description
CCE	The “ConsolidateConditionalExpression” refactoring combines them into a single conditional expression and extracts it.
CDCF	The “ConsolidateDuplicateConditionalFragments” refactoring moves it outside of the expression.
RCF	The “RemoveControlFlag” refactoring uses a break or return instead of a variable that is acting as a control flag for a series of boolean expressions.
RNCwGC	The “ReplaceNestedConditionalwithGuardClauses” refactoring uses guard clauses for all the special cases.
RCwP	The “ReplaceConditionalwithPolymorphism” refactoring moves each leg of the conditional to an overriding method in a subclass. Make the original method abstract.
INO	The “IntroduceNullObject” refactoring replaces the null value with a null object.
IA	The “IntroduceAssertion” refactoring makes the assumption explicit with an assertion.
RM	The “RenameMethod” refactoring transforms a name of the method to a new one.
AP	The “AddParameter” refactoring adds a parameter to one method.
RP	The “RemoveParameter” refactoring removes a parameter from the method.
SQfM	The “SeparateQueryfromModifier” refactoring replaces one method that returns a value but also changes the state of an object by create two methods, one for the query and one for the modification.
PM	The “ParameterizeMethod” refactoring creates one method that uses a parameter for the different values to replaces several methods.
RPwEM	The “ReplaceParameterwithExplicitMethods” refactoring creates a separate method for each value of the parameter in the method.
PWO	The “PreserveWholeObject” refactoring sends the whole object instead of several parameters from this object.
RPwM	The “ReplaceParameterwithMethods” refactoring removes parameter and invokes the method.
IPO	The “IntroduceParameterObject” refactoring replaces parameters with an object.
RSM	The “RemoveSettingMethod” refactoring removes any setting method for that field.
HM	The “HideMethod” refactoring makes the method private.
RCwF	The “ReplaceConstructorwithFactoryMethod” refactoring replaces the constructor with a factory method.
ED	The “EncapsulateDowncast” refactoring moves the downcast to within the method.
RECwE	The “ReplaceErrorCodewithException” refactoring throws an exception instead of a special code which indicates an error.
REwT	The “ReplaceExceptionwithTest” refactoring changes the caller to make the test first.

next page...

RefOps	Transformation Description
PUF	The “PullUpField” refactoring moves the field to the superclass from the subclasses.
PUM	The “PullUpMethod” refactoring moves the method to the superclass from the subclasses.
PUCB	The “PullUpConstructorBody” refactoring creates a superclass constructor; call this from the subclass methods.
PDM	The “PushDownMethod” refactoring moves the method to the subclasses from the superclass.
PDF	The “PushDownField” refactoring moves the field to the subclasses from the superclass.
ESubc	The “ExtractSubclass” refactoring creates a subclass for that subset of features.
ESuperc	The “ExtractSuperclass” refactoring creates a superclass and moves the common features to the superclass.
EI	The “ExtractInterface” refactoring extracts the subset into an interface.
CH	The “CollapseHierarchy” refactoring merges superclass and subclass together
FTM	The “FormTemplateMethod” refactoring transforms the original methods to template methods and pulls them up to superclass.
RIwD	The “ReplaceInheritancewithDelegation” refactoring creates a field for the superclass, adjusts methods to delegate to the superclass, and removes the subclassing.
RDwI	The “ReplaceDelegationwithInheritance” refactoring makes the delegating class a subclass of the delegate.

[1]The transformation descriptions of each operators are derived from the book[Fow99].

Appendix B

Table B.1: Optimization Rules

Former Refactoring	Latter Refactoring	Merged Refactoring
$\mathrm{RF}_{C.f1 \Rightarrow C.f2}$	$\mathrm{RF}_{C.f2 \Rightarrow C.f3}$	$\mathrm{RF}_{C.f1 \Rightarrow C.f3}$
$\mathrm{RF}_{C.f1 \Rightarrow C.f2}$	$\mathrm{IT}_{C.f2 \Rightarrow C.expression}$	$\mathrm{IT}_{C.f1 \Rightarrow C.expression}$
$\mathrm{RF}_{C1.f1 \Rightarrow C1.f2}$	$\mathrm{RTwQ}_{C1.f2 \Rightarrow C1.M}$	$\mathrm{RTwQ}_{C1.f1 \Rightarrow C1.M}$
$\mathrm{IEV}_{expression \Rightarrow C.f1}$	$\mathrm{RF}_{C.f1 \Rightarrow C.f2}$	$\mathrm{IEV}_{C.expression \Rightarrow C.f2}$
$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{RM}_{C.M2 \Rightarrow C.M3}$	$\mathrm{RM}_{C.M1 \Rightarrow C.M3}$
$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{IM}_{C.M2 \Rightarrow \ldots}$	$\mathrm{IM}_{C.M1 \Rightarrow \ldots}$
$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{EC}_{C.M2 \Rightarrow C.M3 \mid C.M4}$	$\mathrm{EC}_{C.M1 \Rightarrow C.M3 \mid C.M4}$
$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{SQfM}_{C.M2 \Rightarrow C.M3 \mid C.M4}$	$\mathrm{SQfM}_{C.M1 \Rightarrow C.M3 \mid C.M4}$
$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{RSM}_{C.M2 \Rightarrow null}$	$\mathrm{RSM}_{C.M1 \Rightarrow null}$
$\mathrm{EM}_{\ldots \Rightarrow C.M2}$	$\mathrm{RM}_{C.M2 \Rightarrow C.M3}$	$\mathrm{EM}_{\ldots \Rightarrow C.M3}$
$\mathrm{IFM}_{\ldots \Rightarrow C.M2}$	$\mathrm{RM}_{C.M2 \Rightarrow C.M3}$	$\mathrm{IFM}_{\ldots \Rightarrow C.M3}$
$\mathrm{EC}_{C.M \Rightarrow CM1 \mid C.M2}$	$\mathrm{RM}_{C.M2 \Rightarrow C.M3}$	$\mathrm{EC}_{C.M \Rightarrow CM1 \mid C.M3}$
$\mathrm{SQfM}_{C.M \Rightarrow CM1 \mid C.M2}$	$\mathrm{RM}_{C.M2 \Rightarrow C.M3}$	$\mathrm{SQfM}_{C.M \Rightarrow CM1 \mid C.M3}$
$\mathrm{PM}_{C.M1 \mid C.M2 \mid \ldots \Rightarrow C.M0}$	$\mathrm{RM}_{C.M0 \Rightarrow C.M}$	$\mathrm{PM}_{C.M1 \mid C.M2 \mid \ldots \Rightarrow C.M}$
$\mathrm{RPwEM}_{C.M \Rightarrow C.M \mid C.M1 \mid C.M2 \mid \ldots}$	$\mathrm{RM}_{C.M1 \Rightarrow C.M0}$	$\mathrm{RPwEM}_{C.M \Rightarrow C.M \mid C.M0 \mid C.M2 \mid \ldots}$
$\mathrm{RCwFM}_{C.constructor \Rightarrow C.factoryM}$	$\mathrm{RM}_{C.factoryM \Rightarrow C.factoryM1}$	$\mathrm{RCwFM}_{C.constructor \Rightarrow C.factoryM1}$
$\mathrm{IPO}_{C.M \Rightarrow C1}$	$\mathrm{RC}_{C1 \Rightarrow C2}$	$\mathrm{IPO}_{C.M \Rightarrow C2}$
$\mathrm{RC}_{C1 \Rightarrow C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{RC}_{C1 \Rightarrow C3}$
$\mathrm{RC}_{C1 \Rightarrow C2}$	$\mathrm{IC}_{C \mid C2 \Rightarrow C}$	$\mathrm{IC}_{C \mid C1 \Rightarrow C}$
$\mathrm{RC}_{C1 \Rightarrow C2}$	$\mathrm{CH}_{C \mid C2 \Rightarrow C}$	$\mathrm{CH}_{C \mid C1 \Rightarrow C}$
$\mathrm{ExtractC}_{\ldots \Rightarrow C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{Extract}_{\ldots \Rightarrow C3}$
$\mathrm{ILE}_{C1 \Rightarrow C1 \mid C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{ILE}_{C1 \Rightarrow C1 \mid C3}$
$\mathrm{RAwO}_{C.f[3] \Rightarrow C \mid C1 \mid C1.f1 \mid C1.f2 \mid C1.f3}$	$\mathrm{RC}_{C1 \Rightarrow C2}$	$\mathrm{RAwO}_{C.f[3] \Rightarrow C \mid C2 \mid C2.f1 \mid C2.f2 \mid C2.f3}$
$\mathrm{ExtractSubclass}_{C1 \Rightarrow C1 \mid C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{ExtractSubclass}_{C1 \Rightarrow C1 \mid C3}$
$\mathrm{ExtractSuperclass}_{C \mid C1 \Rightarrow C \mid C1 \mid C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{ExtractSuperclass}_{C \mid C1 \Rightarrow C \mid C1 \mid C3}$
$\mathrm{EI}_{C1 \Rightarrow C1 \mid I}$	$\mathrm{RC}_{I \Rightarrow I1}$	$\mathrm{EI}_{C1 \Rightarrow C1 \mid I1}$
$\mathrm{DOD}_{C1 \Rightarrow C1 \mid C2}$	$\mathrm{RC}_{C2 \Rightarrow C3}$	$\mathrm{DOD}_{C1 \Rightarrow C1 \mid C3}$
$\mathrm{RTwQ}_{C.f \Rightarrow C.M1}$	$\mathrm{RM}_{C.M1 \Rightarrow C.M2}$	$\mathrm{RTwQ}_{C.f \Rightarrow C.M2}$
$\mathrm{SEF}_{C.f \Rightarrow C.f \mid C.M1 \mid C.M2}$	$\mathrm{RM}_{C.M1 \Rightarrow C.M3}$	$\mathrm{SEF}_{C.f \Rightarrow C.f \mid C.M3 \mid C.M2}$
$\mathrm{EF}_{C.f \Rightarrow C.f \mid C.M1 \mid C.M2}$	$\mathrm{RM}_{C.M1 \Rightarrow C.M3}$	$\mathrm{EF}_{C.f \Rightarrow C.f \mid C.M3 \mid C.M2}$
$\mathrm{RDVwO}_{C.f \Rightarrow C \mid NewClass \mid NewClass.f}$	$\mathrm{RC}_{NewClass \Rightarrow C2}$	$\mathrm{RDVwO}_{C.f \Rightarrow C \mid C2 \mid C2.f}$

next page. . .

Former Refactoring	Latter Refactoring	Merged Refactoring
RRwDC$_{Record \Rightarrow C1}$	RC$_{C1 \Rightarrow C2}$	INO$_{Record \Rightarrow C2}$
RTCwC$_{C1.f1 \mid C1.f2 \mid C1.f3 \Rightarrow C1 \mid C2.f1 \mid C2.f2 \mid C2.f3}$	RC$_{C2 \Rightarrow C3}$	RTCwC$_{C1.f1 \mid C1.f2 \mid C1.f3 \Rightarrow C1 \mid C3.f1 \mid C3.f2 \mid C3.f3}$
RTCwS$_{C1 \mid C1.f1 \mid C1.f2 \Rightarrow C1 \mid C2 \mid C3 \mid C2.f1 \mid C3.f2}$	RC$_{C2 \Rightarrow C4}$	RTCwS$_{C1 \mid C1.f1 \mid C1.f2 \Rightarrow C1 \mid C4 \mid C3 \mid C4.f1 \mid C3.f2}$
RTCwSS$_{C1.f1 \mid C1.f2 \Rightarrow C1 \mid C2.f1 \mid C3.f2}$	RC$_{C3 \Rightarrow C4}$	RTCwSS$_{C1.f1 \mid C1.f2 \Rightarrow C1 \mid C2.f1 \mid C4.f2}$
INO$_{...NULL... \Rightarrow C1}$	RC$_{C1 \Rightarrow C2}$	INO$_{...NULL... \Rightarrow C2}$
RMNwSC$_{C.NUMBER \Rightarrow C.f1}$	RF$_{C.f1 \Rightarrow C.f2}$	RMNwSC$_{C.NUMBER \Rightarrow C.f2}$

Appendix C

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$[1]
1	12828	10328	10890	16	15	63	10984
2	11875	10297	8609	16	31	63	8719
3	11906	10281	8594	0	47	47	8688
4	11968	10453	8625	0	47	63	8735
5	12093	5891	8610	15	47	47	8719
6	11828	10282	8625	0	31	47	8703
7	11844	10266	8750	15	63	47	8875
8	11953	10234	8516	16	31	47	8610
9	11938	10281	8547	0	47	47	8641
10	11953	10391	8578	16	47	31	8672
$T_{average}$	12018.6	9870.4	8834.4	9.4	40.6	50.2	8934.6

Table C.1: Detailed data of 5-RFM sequence#1 in "SimplifiedList" case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	12828	10296	9688	0	93	125	9906
2	12875	10469	9219	0	46	63	9328
3	12890	10406	9328	16	62	47	9453
4	12813	10515	9203	0	62	63	9328
5	12844	5984	9187	16	62	47	9312
6	12813	10328	9234	16	47	62	9359
7	12875	10578	9157	15	47	63	9282
8	12766	5954	9203	16	46	63	9328
9	12859	10328	9234	16	46	47	9343
10	12844	10610	9266	0	47	62	9375
$T_{average}$	12840.7	9546.8	9271.9	9.5	55.8	64.2	9401.4

Table C.2: Detailed data of 8-RFM sequence#2 in "SimplifiedList" case

[1] "No.": test's times; "$T_{unoptimized}$": the compilation time of composing unoptimized sequence; "$T_{optimized}$": the compilation time of composing optimized sequence; "T_1": the compilation time of loading RFMs into the optimization tool; "T_2": the compilation time of optimizing RFMs; "T_3": the compilation time of rewriting RFMs and equation file; "T_4": the compilation time of deleting temp folders; "$T_{optimization}$": the compilation time of total optimization process; "ms": millisecond.

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	16625	10422	10422	62	47	62	10593
2	16375	10344	9703	15	32	62	9812
3	16344	10343	9687	16	31	63	9797
4	16359	10547	9843	16	47	62	9968
5	16390	10343	5250	16	31	62	5359
6	16390	10484	9812	16	125	62	10015
7	16266	10312	9734	16	46	63	9859
8	16281	10578	9781	16	93	63	9953
9	16266	10391	9891	15	16	62	9984
10	16297	10359	5297	15	32	62	5406
$T_{average}$	16359.3	10412.3	8942	20.3	50	62.3	9074.6

Table C.3: Detailed data of 10-RFM sequence#3 in “SimplifiedList” case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	32657	14156	10110	15	47	63	10235
2	31797	14140	5359	16	31	78	5484
3	31875	14094	9828	15	32	62	9937
4	31843	14094	5344	16	47	62	5469
5	31797	13984	9938	15	16	78	10047
6	31843	14188	5391	16	31	62	5500
7	31828	14016	9796	32	31	78	9937
8	32015	14062	5328	31	32	78	5469
9	31828	14046	9812	16	94	78	10000
10	31859	14156	9875	16	31	63	9985
$T_{average}$	31934.2	14093.6	8078.1	18.8	39.2	70.2	8206.3

Table C.4: Detailed data of 10-RFM sequence in “TankWar” case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	176063	83328	18453	16	31	375	18875
2	168500	79469	18312	31	32	328	18703
3	169235	100375	16938	16	46	313	17313
4	175218	97984	18812	16	47	312	19187
5	162609	79313	18157	15	47	328	18547
6	175609	79547	18907	15	47	313	19282
7	166375	79094	18406	16	31	312	18765
8	187578	79797	18781	16	47	312	19156
9	173375	78157	18203	16	47	343	18609
10	167062	78547	18687	15	32	328	19062
$T_{average}$	172162.4	83561.1	18365.6	17.2	40.7	326.4	18749.9

Table C.5: Detailed data of 10-RFM sequence#1 in “workbench.texteditor” case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	250078	63375	18531	31	32	203	18797
2	247265	59078	18266	15	32	203	18516
3	244375	59625	17672	15	32	234	17953
4	251359	59782	18516	31	32	203	18782
5	274938	59453	18047	31	31	297	18406
6	275578	58922	18359	16	31	203	18609
7	254813	59031	18187	16	47	234	18484
8	247625	59453	18578	16	31	203	18828
9	248328	58968	17906	31	32	203	18172
10	243953	59625	17687	31	16	203	17937
$T_{average}$	253831.2	59731.2	18174.9	23.3	31.6	218.6	18448.4

Table C.6: Detailed data of 17-RFM sequence#2 in "workbench.texteditor" case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	753422	73422	76640	94	47	1531	78312
2	769609	58735	77093	94	47	1531	78765
3	802047	59156	71813	109	31	1485	73438
4	747875	59359	77297	109	328	1578	79312
5	754844	58656	73906	94	63	1687	75750
6	736412	60032	77516	109	344	1609	79578
7	810604	59968	75828	94	109	1516	77547
8	802047	59062	76094	109	375	1657	78235
9	745472	65109	76297	109	47	1531	77984
10	773843	59422	75422	93	47	1844	77406
$T_{average}$	769617.5	61292.1	75790.6	101.4	143.8	1596.9	77406

Table C.7: Detailed data of 55-RFM sequence#3 in "workbench.texteditor" case

No.	$T_{unoptimized} ms$	$T_{optimized} ms$	$T_1 ms$	$T_2 ms$	$T_3 ms$	$T_4 ms$	$T_{optimization} ms$
1	20891	20203	10344	31	31	250	10656
2	20860	20265	8438	16	31	94	8579
3	20453	20281	8375	0	93	78	8546
4	20391	20344	8438	15	47	78	8578
5	20203	20282	8438	0	47	78	8563
6	20391	20313	3922	0	31	78	4031
7	20422	20360	8406	0	47	78	8531
8	20281	20265	3953	15	16	78	4062
9	20328	20266	8438	15	31	79	8563
10	20390	20235	8437	16	16	93	8562
$T_{average}$	20461	20281.4	7718.9	10.8	39	98.4	7867.1

Table C.8: Detailed data of 3-RFM sequence in "ZipMe" case

Selbstststädigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, March 1, 2010

Liang Liang