UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

TABLE OF CONTENTS

LIST OF FIGURES

Page

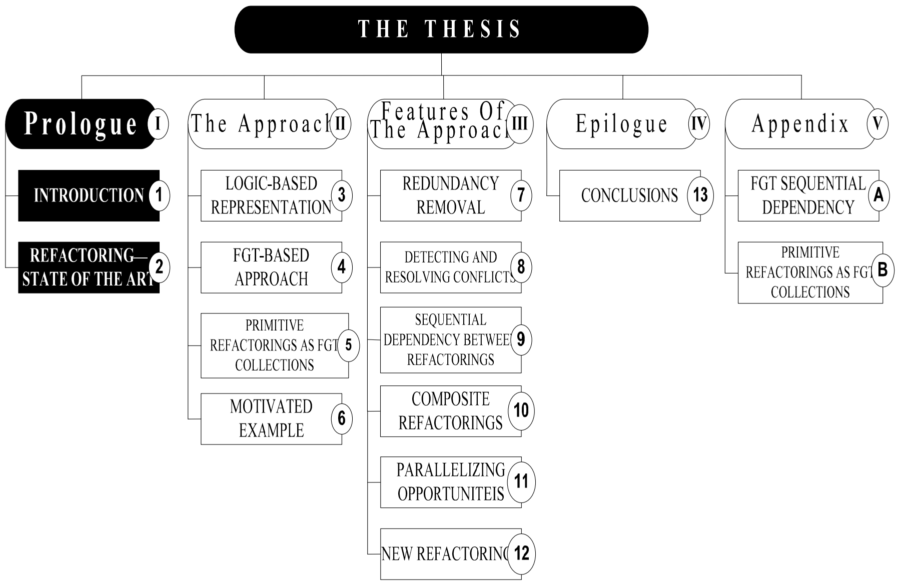
LIST Of TABLES

Page

LIST OF ALGORITHMS

Part I

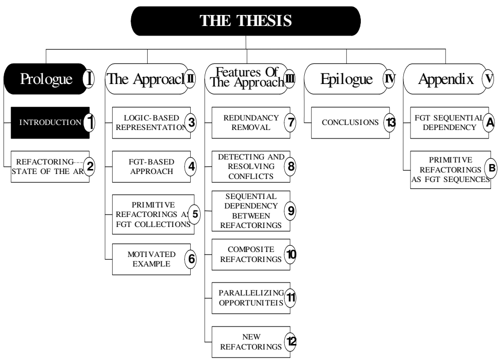
Prologue


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING—STATE OF THE ART 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORING 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

CHAPTER 1

INTRODUCTION

1.1 The Problem


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
The Approacl II
LOGIC-BASED REPRESENTATIO 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT SEQUENCES B

Software that is used in a real-world environment inevitably changes or becomes progressively less useful in that environment. As evolving software changes, its structure tends to become more complex [46]. "Because of this, the major part of the total software development cost is devoted to software maintenance [9, 29, and 47]. Better software development methods and tools do not solve this problem, because their increased capacity is used to implement more new requirements within the same time frame [25], making the software more complex again. To cope with this spiral of complexity, there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality. The research domain that addresses this problem is referred to as restructuring [1, 28] or, in the specific case of object-oriented software development, refactoring [22, 65]." [59]

Refactoring is the process of improving the internal structure of the software while preserving its external behaviour [22, 65 and 70]. By improving the internal structure it is meant that refactoring will restructure the software in order to improve its quality by making it easier to understand, to extend, to find bugs, and to program faster [2, 60]. Preserving the external behaviour means, before and after applying the refactoring, the software will require the same preconditions and result in the same postconditions. The refactoring community assumes a set of precondition conjuncts for each refactoring that needs to be satisfied as a condition for applying that refactoring.

To give an idea about refactoring before going into the details of the thesis, Figure 1.1(a) shows a simple example of a UML class diagram with four classes: *HR*, *Employee* as a superclass, *Salesman* and *Engineer* as subclasses of *Employee*. The *HR* class has two association relations, one with each of the *Salesman* and *Engineer* classes. The *Salesman* and *Engineer* subclasses have the same method *getName* which is called by the method *report* in the *HR* class.

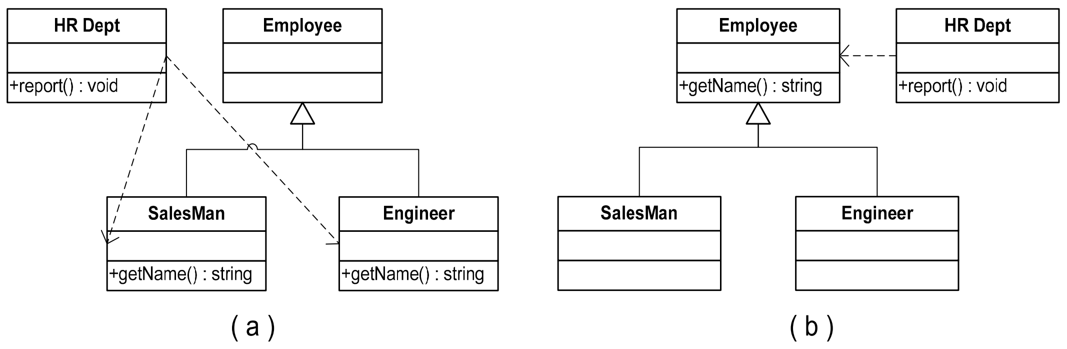


Figure 1.1: pullUpMethod Refactoring: (a) before refactoring, (b) after refactoring

Note that the duplication of the *getName* method in the two subclasses as shown in Figure 1.1(a) causes the following design problems:

1. More efforts and spaces are needed at the design and code levels.

2. There is an increased chance of inconsistency between the two copies. This can arise if the developer changes one of the two copies and forgets to change the other.

3. The design is complicated, because the same method appears two times in the design. This also causes two association relations to be created between *HR* class and each one of the two subclasses.

To solve these problems, it is preferred to change the design by deleting the *getName* method from the two subclasses and move it to their superclass, as shown in Figure 1.1(b). As a result, one copy of the *getName* method will appear in the design and also the two association relations between the *HR* class and the two subclasses will be replaced by one association relation between the *HR* class and the *Employee* class. Doing this restructuring will increase the quality of the internal design of the class diagram without changing the external behaviour of the system (The system will make the same services as before restructuring). This is because the *getName* method will be inherited to the two subclasses. The associated association relation also will be inherited.

The restructuring done in the previous example is an example of refactoring. In this case, it is a **pullUpMethod** refactoring. The precondition for the **pullUpMethod** refactoring that should be satisfied in order to apply the refactoring to the system, as a condition to preserve the behaviour of the system is:

1. The *getName* method should not be declared in the superclass (*Employee*) or any of its ancestors.

2. The access mode of the *getName* method in the subclasses is not *private.*

3. All the references made by the *getName* method must be visible from the superclass.

4. The signature of the method in all the subclasses should be the same.

A current research trend is to investigate refactorings at levels of abstraction above the code-level [23, 68, and 81]. This is because many people are visually oriented and prefer to visualize the relationships between classes rather than apprehend them textually. Furthermore, being able to directly manipulate code at a higher level of granularity (i.e. methods, variables, and classes rather than characters) can make refactoring more efficient [2]. Therefore, this thesis also focusses on refactorings at the design level.

Several approaches have been used to formalize such refactorings, as discussed in section 2.4. For example, the graph transformations approach [11, 18 and 19] represents software as a graph, and refactorings are formalized as graph-production rules [7, 34, 51-56, and 63]. As another approach, the logic-based conditional transformation approach [38, 39] represents software as logic-terms and refactorings are formalized as conditional transformations with pre- and postconditions.

In general, reasoning takes place at the level of refactorings themselves, and attention is not paid to the detailed transformational steps that must be applied to the model to achieve the refactoring. Such reasoning is with respect to a set of preconditions that must be satisfied in order to apply that refactoring, resulting in a set of postconditions. In this sense, a refactoring is treated as an abstraction, or as a black box as illustrated in Figure 1.2.

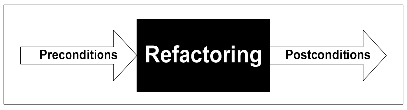


Figure 1.2: Refactorings as black box

Of course, to be of practical value, these conceptual ideas have to be implemented in refactoring tools. Such a tool would have to access some representation of an underlying system that is to be refactored. The refactorings themselves are implemented as hard coded parameterise procedures—i.e. as a sequence of code statements. To apply a particular refactoring to the underlying system, the tool requires an interface that allows the user to select and invoke procedures which then execute the actual refactoring, thus changing the underlying system representation accordingly.

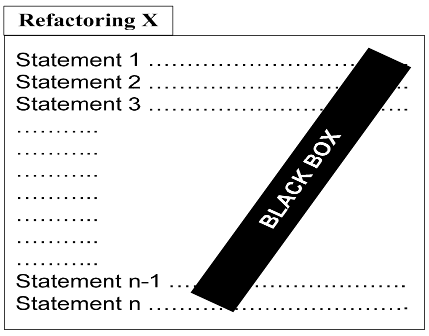
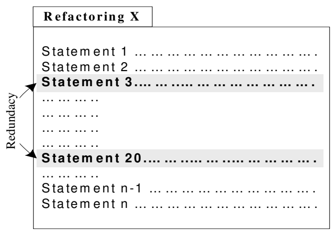


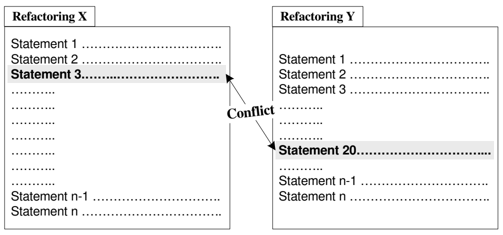
Figure 1.3: Refactorings as hard coded sequence of statements

Treating refactoring as a black box can be notionally conceived of as shown in Figure 1.3. Whenever a refactoring is applied, the hard coded sequence of statements is executed atomically. The inter-relationship between the different code statements both within and between refactorings cannot be determined. This has the following implications:

1. Where redundancy inside or between refactoring may exist, there is no possibility to remove it. As shown in the figure on the right, there could be a redundancy between statement 3 and 20 in the code. For example, if statement 3 adds an attribute to a specific class in the system and subsequently statement 20 deletes or changes the name or definition type of that attribute, the redundancy cannot be removed. This kind of scenario could arise, for example, when composing two or more refactorings into a single one.

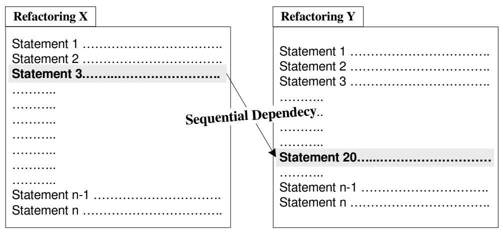


2. Where conflict occurs between two refactorings, it is not possible to determine which part of the two refactorings caused the conflict. The figure on the right side illustrates this by showing a conflict between statement 3 in refactoring X and statement 20 in refactoring Y. For example, statement 3 might add an attribute to a specific class in the system, based on a precondition of refactoring X that the class exists but does not have that attribute. On the other hand, statement 20 might delete that class from the system, based on the precondition—that the class exists and has no attributes.

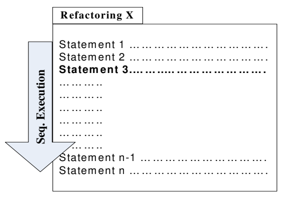


This would constitute a conflict between the two refactorings if they were to be applied as separate threads to the system.

3. Where there is a sequential dependency between two refactorings, there is no possibility to know at what specific point on it one of the two refactorings is sequentially dependent on the other. As shown in the figure on the right side if there is a sequential dependency between statement 3 in refactoring X and statement 20 in refactoring Y. Where statement 3, for example, adds a class to the system and statement 20 adds an attribute to that class. In this case refactoring Y is considered to be sequentially dependent on refactoring X and having to be applied to the system after refactoring X. Again, because the two refactorings are considered as code sequences, there is no possibility to know at what specific point in the code one of the two refactorings becomes sequentially dependent on the other.



4. Because refactorings are considered as code sequences, two or more refactorings can only be run in parallel if they are shown to be sequentially independent of each other. Because there is no meta-information about the nature of sequential dependency between their constituent code statements, it is not possible to determine whether parts of the refactorings could be run in parallel.



5. A new composite refactoring can be assembled by using previously-defined refactorings as building blocks. Its constituent elements can only be analysed for redundancy, conflicts, sequential dependency and possible parallelization with reference to the pre- and postconditions of these elements—i.e. with reference to the properties of the original refactorings. Nevertheless, as will be discussed later, such an analysis can suggest an ordering of the constituent refactorings which will avoid the so-called rollback problem.

6. If a tool allows a user to build new refactorings, the semantics of any new refactoring is necessarily constrained by the selection of refactorings that have been implemented in the tool. Any refactoring whose semantics goes beyond that will have to be hard coded as a task to be undertaken by the tool developer, rather than the tool user.

1.2 The Proposed Formalism

The refactoring formalism proposed in this thesis and described briefly in [73-75], is based on a predefined set of fine-grain transformations (FGTs) which are the basis for the construction of refactorings. These FGTs are derived from the general transformation actions that can be performed on elements of a UML class model. Each FGT can be applied to a UML model of a system, provided that the system satisfies the FGT's precondition. The FGT's postcondition is then realized on the system, which represents, in general, a small incremental change to the system. Note that this change need not preserve system behaviour.

Nevertheless, it will be shown that refactorings (which, of course, do preserve system behaviour) can be constructed by using a collection of these FGTs. As illustrated in Figure 1.4, a set of refactorings in the present approach is set of directed acyclic graphs (FGT-DAGs), each of which specifies an ordering of FGTs to be used in the refactoring. The order, effect, pre- and postcondition of each FGT in each FGT-DAG is known to the tool, and can be controlled at the time of refactoring. Of course, the final effects of refactoring X in Figure 1.4 is the same as the final effects of a hard coded version of refactoring X in Figure 1.3.

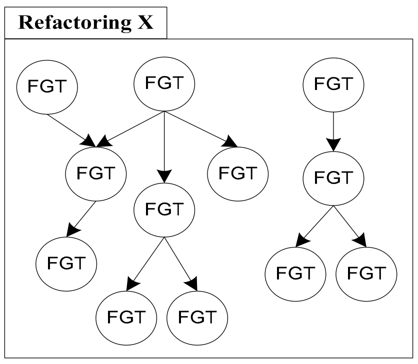


Figure 1.4: Refactoring as a set of FGT-DAGs

It will be shown that representing refactorings as a collection of FGTs allows for the following:

1. Where redundancy occurs between transformation operations that are carried out by the refactorings, the redundancy can be discovered and removed at the FGT-level. (This will be discussed in more detail in chapter 7)

2. In the case of conflict between two refactorings, the FGTs that cause the conflict can be discovered, and in some cases the conflict can be resolved without withdrawing one of the refactorings. (This will be discussed in more detail in chapter 8)

3. Sequential dependency between two refactorings can be discovered at the FGT-level. (This will be discussed in more detail in chapter 9)

4. Composite refactorings of more than one refactoring can be composed in a way that will avoid rollback problems. However, this is done by manipulating the ordering of FGT execution, rather than of refactoring execution. (This will be discussed in more detail in chapter 10)

5. Parallel execution can be exploited at the FGT-DAG level. Thus, all FGT-DAGs in one refactoring can be executed concurrently because there is no sequential dependency between the FGT-DAGs. For example, the refactoring in Figure 1.4 has two FGT-DAGs that can be manipulated concurrently. (This will be discussed in more detail in chapter 11)

6. An FGT-based tool can be built that will allow a user to build new refactorings whose semantics is constrained, not by the selection of existing refactorings that have been implemented in the tool, but rather by the semantics of the FGTs that have been predefined in the tool. (This will be discussed in more detail in chapter 12).

The discussion in this thesis is restricted to refactorings that relate to the simplified UML meta-model shown in Figure 1.5. In addition, it will be assumed that a limited amount of information derived from the source code of the system to be refactored is also available, as will be discussed in due course. Although the use of this code-based information goes beyond the requirements of existing approaches, it can be acquired fairly easily.

In deciding of which features of UML to include and which to exclude from the study, consideration had to be given to having a subset of the UML vocabulary that would be sufficiently large to lend credibility to the approach, yet not be so ambitious that it would prevent full coverage within the time available for this study. It was thought that the vocabulary represented by the simplified meta-model of FIgure 1.5 complied with this objective. Although, UML notations relating to interfaces, abstract classes, abstract methods, aggregations and so on, are not considered, extending the ideas developed in this thesis to these UML notations appears to be quite straightforward. However, a detailed investigation of this conjecture is a matter for future study.

It should be noted there are tools (such as IDEA by IntelliJ and Eclipse by IBM) that directly analyse and manipulate an existing code base. However, the types of refactorings that they address are generally of a different order to those addressed here (e.g. removal of declared but unused variables, or the identification of common code segments that can be turned into a method) and are beyond the scope of this thesis.

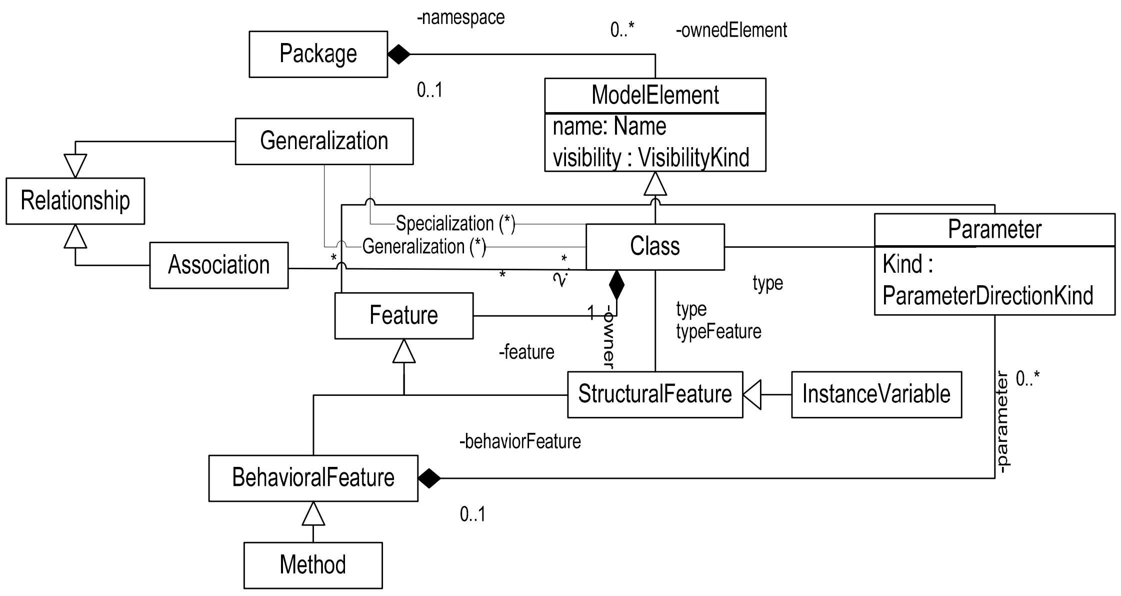


Figure 1.5: Simplified UML meta-model

1.3 Thesis Overview

In the next chapter, a survey of previous work in refactoring is presented. Thereafter, chapters three to six present the proposed approach and discuss the feasibility of the approach for formalizing refactorings. Then, chapters seven to twelve discuss the features that are obtained by adopting such approach.

The logic-based underlying representation of the UML class diagrams of the system under consideration is presented in chapter 3. Chapter 4 proposes an FGT-based methodology to construct model transformations in which FGTs are at the core of the refactoring system. Several common primitive refactorings that are frequently defined and used in the refactoring literature are presented in chapter 5. To illustrate the proposed approach, a motivated example is given in chapter 6.

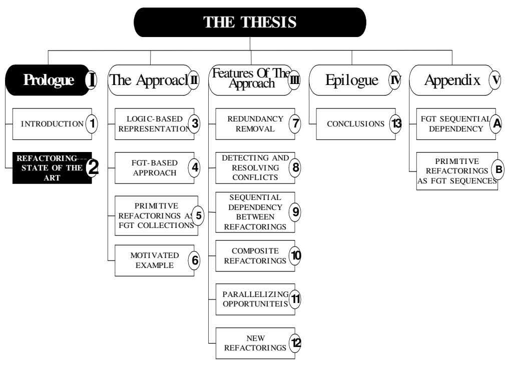
Presenting features of the approach is started in chapter 7. The chapter introduces the idea of removing the redundancy between FGTs allocated in the same FGT-DAG. Chapter 8 shows how to detect and resolve conflicts that may occur between two refactorings. The sequential dependency between two refactorings is discussed in chapter 9. Chapter 10 discussed the implications of using FGTs to deal with composite refactorings. The opportunities for parallelizing refactorings are presented in chapter 11. Chapter 12 presents the possibility for end users to build their own refactorings. Finally, chapter 13 summarizes the work, explores the contributions and identifies tasks for future work.

In summary, then, this thesis will show that when FGTs are used to build refactorings, all the well-known refactoring operations (such as determining redundancy, conflict and sequential dependency; and building composites) can take place at the FGT-level. In theory, this comes with certain advantages and disadvantages. Advantages include the fact that the user of an FGT-based tool will have enhanced flexibility in specifying new refactorings; redundancies and conflicts can be more accurately pin-pointed and removed; and opportunities for parallel execution are exposed at a more fine-grained level. It will be seen that these advantages come at the cost of having to carry out more computations because analysis has to take place at the FGT-level, rather than at what will later be called the “refactoring level". Although a prototype tool has been built to verify these claims, the full practical implications of this work are a matter for future study.

CHAPTER 2

REFACTORING — STATE OF THE ART

In this chapter, a survey of work related to refactoring is presented. First, the concept of software evolution and its relation to refactoring is introduced. Then, works related to different types of software artifacts that can be refactored is presented. Finally, works related to different refactoring formalisms is discussed.


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT SEQUENCES B

2.1 Software Evolution

“Software evolution is an essential part of the software development process. Nearly all software inevitably undergoes changes during its lifetime. Changes can be large or small, simple or complex, important or trivial - all of which influence the effort needed to implement the changes“ [51]. Sommerville [79] explains that proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system’s lifetime. Lehman & Belady [46] conducted empirical studies into software evolution and concluded the following eight laws:

1. Continuing change: Software that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

2. Increasing complexity: As evolving software changes, its structure tends to become more complex. Extra resources must be devoted to preserve and simplify the structure.

3. Large program evolution: Software evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.

4. Organizational stability: Over a software lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

5. Conservation of familiarity: Over the lifetime of a software, the incremental change in each release is approximately constant.

6. Continuing growth: The functionality offered by systems has to continually increase to maintain user satisfaction.

7. Declining quality: The quality of systems will appear to be declining, unless they are adapted to changes in their operational environment.

8. Feedback system: Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Experience over the last 30 years has shown that making software changes without visibility into their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system. The immaturity of current-day software evolution is clearly stated in the foreword of the international workshop on principles of software evolution [69]:

"*Software evolution is widely recognised as one of the most important problems in software engineering. Despite the significant amount of work that has been done, there are still fundamental problems to be solved. This is partly due to the inherent difficulties in software evolution, but also due to the lack of basic principles for evolving software systematically.*"

Software evolution is not restricted to the implementation phase only. Even in the earlier phases of requirements specification, analysis and design, evolution is a strict necessity. To date, most research on evolution has been dedicated to the implementation and maintenance phases, and to a lesser degree in the earlier phases of requirements specification and design [12, 15, 33, 41, 87, and 88]. However, there is a tendency to shift towards earlier phases.

2.2 Refactoring

Although in the context of *software reengineering*, refactoring is often used to convert legacy code into a more modular or structured form [20], refactoring can also be applied to any type of software artifact. For example, it is possible and useful to refactor design models, database schemas, software architectures and software requirements. Refactoring of these kinds of software artifacts rids the developer from many implementation-specific details, and raises the expressive power of the changes that are made. On the other hand, applying refactorings to different types of software artifacts introduces the need to keep them all in sync[59].

In the following subsections, an introduction of refactorings at different types of software artifacts is given.

2.2.1 Codes Level

2.2.1.1 Non-Object-Oriented Programming Languages

Programs that are not written in an object-oriented language are more difficult to restructure because data flow and control flow are tightly interwoven. Because of this, restructurings are typically limited to the level of a function or a block of code [59].

In [27], Griswold proposes a technique to restructure programs written in a block-structured programming language. The language he worked on is Scheme. His transformations concern program restructuring for aiding maintenance. To insure that the transformations are meaning preserving, he uses Program Dependence Graphs to reason about the correctness of transformation rules.

Lakhotia and Deprez [42] present a transformation called *tuck* for restructuring programs by decomposing large functions into small functions. The transformation breaks large code fragments and tucks them into new functions. The challenge they faced was creating new functions that capture computations that are meaningfully related. There are three basic transformation to tuck functions.

4. Related code is gathered by driving a wedge (which is a program slice bounded with single-entry and a single exit point) into the function.
5. Then the code that is isolated by the wedge is split.
6. Finally, the split code is folded into a function.

These transformations can even create functions from non-contiguous code.

2.2.1.2 Object-Oriented Programming Languages

Opdyke, in his PhD thesis [65] was the first to introduce the term refactoring. His proposed refactorings were in the context of object-oriented programming languages. He identified twenty-three primitive refactorings and gave examples of three composite refactorings. He arrived at his collection of refactorings by observing several systems and recording the types of refactorings that OO programmers applied.

The importance of the achievements of Opdyke is not only the identification of refactorings, but also the definition of the precondition that is required to apply a refactoring to a program without changing its behaviour. For that, he defined for each primitive refactoring a set of precondition conjuncts that would ensure that the refactoring would preserve behaviour.

Roberts, in his PhD thesis [70], improves the work of Opdyke. He gives a definition of refactoring that focuses on their pre- and postcondition conjuncts. The definition of postcondition conjuncts allows the elimination of program analysis that is required within a chain of refactorings. This comes from the observation that refactorings are typically applied in a sequence intended to set up precondition conjuncts for later refactorings.

In his book [22], Fowler presents a catalogue of refactorings. Each refactoring is given a name and short summary that describes it. A motivation describes why the refactoring should be done, a step-by-step description of how to carry out the refactoring and an example.

Back [3] propose a method called *stepwise feature introduction* for software construction. The method is based on incrementally extending the system with a new feature at a time. Introducing a new feature may destroy some already existing features, so the method must allow for checking that old features are preserved.

2.2.2 Design Level Models

A recent research trend is to deal with refactoring at a design level, for example, in the form of UML models [64]. Applying refactoring to models rather than to source code can encompass a number of benefits [23]. Firstly, software developers can simplify design evolution and maintenance, since the need for structural changes can be more easily identified and addressed on an abstract view of the system. Secondly, developers are able to address deficiencies uncovered by model evaluation, improving specific quality attributes directly on the model. Thirdly, a designer can explore alternative decision paths in a cheaper way (although small prototypes may be necessary). An apparent scenario for model refactorings is the incorporation of design patterns into a system's design model [37].

France *et al.* [23] identified two classes of model transformations: vertical and horizontal transformations. Vertical transformations change the level of abstraction, whereas horizontal transformations maintain the level of abstraction of the target model. A model refactoring is an example of horizontal transformation. In contrast, the Model-Driven Architecture (MDA) approach [78], in which abstract models automatically derive implementation-specific models and source code, provides examples of a vertical transformation.

As the idea of refactoring models adds simplicity to software evolution, automatization and behaviour preservation are even more complex issues when dealing with models. Editing a class diagram may be as simple as adding a new line when introducing an association, but such changes must include identifying lines of affected source code, manually updating the source, testing the changes, fixing bugs and retesting the application until the original behaviour is recovered [83]. Methods and tools for partially or even totally removing human interaction in this process are invaluable for the refactoring practice.

Suny'e *et al.* [81] have provided a fundamental paradigm for model refactoring to improve the design of object-oriented applications. They present refactorings of class diagrams and state charts. In order to guarantee behaviour-preserving transformations of state charts, they specify the constraints that must be satisfied before and after the transformation using the OCL at the meta-model level.

Porres [68] implemented refactorings as a collection of transformation rules, which receives one or more model elements as parameters, and performs a basic transformation based on the parameters.

Boger *et al.* [6] present a refactoring browser integrated into a UML tool. They concentrate on the detection of conflicts that may be introduced after refactorings. They classify conflicts as warnings and errors. Warnings indicate that conflicts might cause a side effect. Errors indicate that an operation will cause damage to the model or code. They also address refactoring of state machines, like merging of states and formation of composite states.

Bottoni, Parisi and Taentzer [7] present an approach to maintain the consistency of specification and code after refactoring. They show that some refactorings require modifications in several diagrams at once. To ensure consistency between source code, structural and behavioural models, they use graph transformations.

Astel [2] proposes using an UML tool as an aid in finding smells—a structure in code that suggest the possibility of refactoring—and performing some elaborate refactorings. It is a tool that bases class diagrams directly on code, allowing code manipulation by the direct manipulation of the diagram.

Gorp *et al.* proposed a UML extension to express the pre- and postcondition of source code refactorings using OCL [26]. The proposed extension allows an OCL empowered CASE tool to verify non-trivial pre- and postcondition, to compose sequences of refactorings, and to use the OCL query engine to detect bad code-smells. Such an approach is desirable as a way to refactor designs independent of the underlying programming language.

2.2.3 Database Schemas Level

The main focus of database schemas is on how data should be structured. Therefore, they are ideal candidates for refactoring. In fact, the research area of object-oriented software refactoring originates from the research on how to restructure object-oriented database schemas.

Banerjee and Kim [4] applied refactoring in the context of database schema evolution. They defined a set of schema transformations, which are used for schema evolution and identified a set of invariant properties of an object-oriented schema which must be preserved across schema changes. An example of such an invariant is that attributes of a class, whether defined or inherited, have distinct names.

2.2.4 Software Architectural Level

In [67] Philipps and Rumpe propose a promising approach to deal with refactorings at the software architecture level. In their work, refactoring rules are based directly on the graphical representation of a system architecture. These rules preserve the behaviour specified by the causal relationship between the components.

Another approach is presented by Tokuda and Batory [83]: architectural changes to two software systems are made by performing a sequence of primitive refactorings (81 refactorings in a first case study, 800 refactorings in a second case study).

In [36] Kempen, Chaudron, and Kourie proposed an approach to refactoring at the software architectural level. In their approach, they use a CSP-based formalism to describe the refactoring and they show that the proposed refactorings indeed preserve behaviour of the system.

2.2.5 Software Requirements Level

Restructuring can also be applied at the requirements specifications level. For example, In [72], Russo *et al.* proposed an approach to refactor the requirement specifications of the system. Their proposal is to restructure natural language requirement specifications by decomposing them into a structure of viewpoints. Each viewpoint encapsulates partial requirements of some system components, and interactions between these viewpoints are made explicit. This restructuring approach increases requirement understandings, and facilitates detecting inconsistencies and managing requirement evolutions.

2.3 Formalisms

A wide variety of formalisms have been proposed and used to deal with refactoring.

2.3.1 Graph Transformations

Graph transformation [10, 11, 18, 19, and 63] is one way to deal with restructuring. The software is represented as a graph, and restructuring corresponds to transformation rules. Mens [51] presents the formalization of refactoring using graph rewriting, a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predefined rules that are described in a graph-production which is specified by means of left-hand and right-hand sides. The first one specifies which parts of the initial graph should be transformed, while the last one specifies the result after transformation.

Mens *et al.* use the graph rewriting formalism to prove that refactorings preserve certain kinds of relationships (updates, accesses and invocations) that can be inferred statically from the source code [54]. Bottoni *et al.* describe refactorings as coordinated graph transformation schemes in order to maintain consistency between a program and its design when any of them evolves by means of a refactoring [7]. Heckel [31] uses graph transformations to formally prove the claim (and corresponding algorithm) of Roberts [70] that any set of refactoring postcondition conjuncts can be translated into an equivalent set of precondition conjuncts. Van Eetvelde and Janssens [17] propose a hierarchical graph transformation approach to be able to view and manipulate the software and its refactorings at different levels of detail.

2.3.2 Pre- and Postcondition

A refactoring's definition can be given in terms of an invariant in the form of a pre- and postcondition that should hold before and after the refactoring has been applied. This can form the basis of a lightweight and automatically verifiable means to ensure that the behaviour of the software is preserved by the refactoring.

The use of pre- and postcondition has been suggested repeatedly in research literature as a way to address the problem of behaviour preservation when restructuring or refactoring software artifacts. In the context of object-oriented database schemas (which are similar to UML class diagrams), Banerjee and Kim identified a set of invariants that preserve the behaviour of these schemas [4]. Opdyke adopted this approach to object-oriented programs, and additionally provided precondition conjuncts or enabling conditions for each refactoring [65]. He argued that this precondition preserves the invariants. Roberts used first order predicate calculus to specify these precondition conjuncts in a formal way [70].

The notion of precondition or applicability condition is also available in the formal restructuring approach of Ward and Bennett, using the formal language WSL [86].

2.3.3 Program Slicing

Program slicing [5, 43, and 82] deals with specific kinds of restructurings: function or procedure extraction. These techniques based on system dependence graphs, can be used to guarantee that a refactoring preserves some selected behaviour of interest. Lakhotia and Deprez [42] present a transformation called *tuck* for restructuring programs by decomposing large functions into small functions. The approach breaks large code fragments and tucks them into new functions.

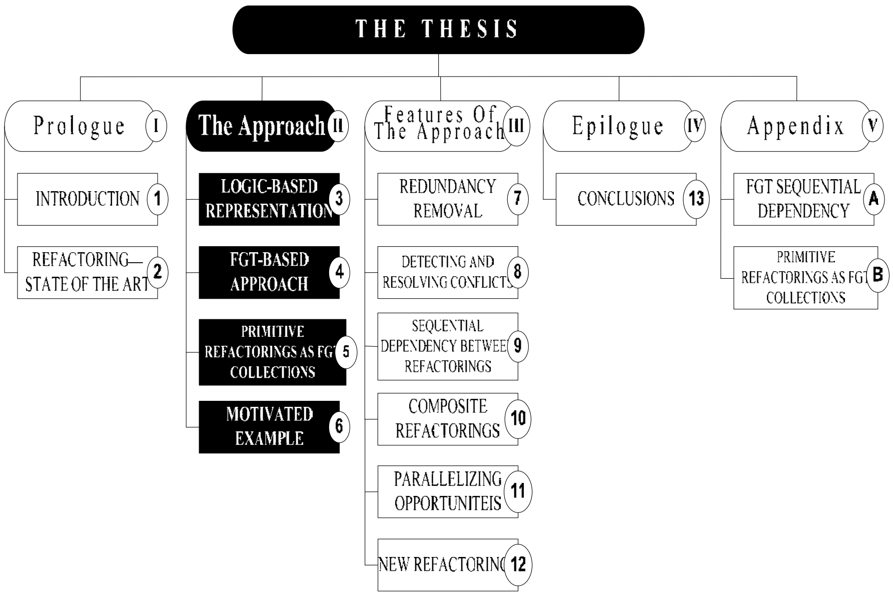
A similar approach is taken in [40], where an algorithm is proposed to move a selected set of nodes in a control flow graph, so that they become extractable while preserving program semantics. They identified conditions based on control and data dependence that are considered to be sufficient to guarantee semantic equivalence.

2.3.4 Formal Concept Analysis

In [24] a technique called *formal concept analysis (FCA)* is used to deal with restructuring. FCA involves clustering so-called objects (not necessarily software objects) according to their attributes. The result is a set of nodes (called concepts) that are hierarchically arranged in a lattice. Snelting in [77] uses FCA to restructure object-oriented class hierarchies. The result is guaranteed to be behaviourally equivalent with the original hierarchy. Tonella in [84] uses the same technique to restructure software modules. Deursen in [14] uses FCA to identify objects by semi-automatically restructuring legacy data structures.

Part II

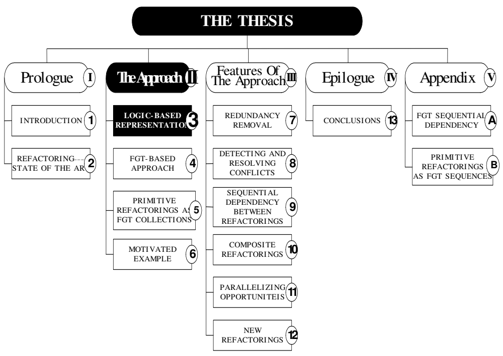
The Approach


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING— STATE OF THE ART 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORING 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

CHAPTER 3

LOGIC-BASED REPRESENTATION

3.1 Introduction


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
The Approach II
LOGIC-BASED REPRESENTATIO 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT SEQUENCES B

In the proposed approach, the core abstract idea is to view refactorings of a system as FGTs, and then to transform the system in terms of these FGTs. As with author mentioned in the previous chapter, the refactoring envisaged here is at the level of the system's design. Ideally, to implement the core idea, a tool would be needed that can make user-requested refactorings on some computer-based description of the system's design. In principle, the tool could be written in any appropriate language, and the representation of the system would therefore have to be designed to match the requirements of that language.

For the purposes of the present study, a prototype tool has been built for experimental purposes. Because of its advanced search engine, and because of its overall suitability for prototyping, it was decided to build the tool in Prolog. A positive consequence of this decision is that many of the forthcoming explanations about the approach can be given by referring to the logic-terms that have been used as data for the Prolog prototype tool.

Moreover, it has been assumed that the system design is represented in standard UML [64]. The first challenge, therefore, is to represent the relevant elements of a UML class diagram as logic-terms. These logic-terms express the semantics of the standard UML modeling vocabulary. The vocabulary consists of a set of *objects* (packages, classes, attributes, methods and parameters) to represent discrete concepts in a class diagram. The vocabulary also contains a set of *relations* (extends, associations, reads, writes, calls, types) to relate the *object* elements in the UML class diagram to one another. The *object* and *relation* elements of concern here are related to the simplified UML meta-model shown in Figure 1.5. Extending the approach to represent other elements in the UML class diagram is straightforward.

In [35], a software refactoring tool called JTRANSFORMER is proposed. The tool represents the full detail of Java code as Prolog facts, and then executes refactorings by manipulating these facts. The inspiration for representing relevant elements of UML class diagrams as logic-terms is based on the concepts described in the JTRANSFORMER tool.

It should be noted, however, that the information required to implement the full range of refactorings mentioned in the literature is not fully available in the UML class diagrams alone. Some refactorings require, in addition, basic *access-related* information—i.e. information that indicates *call* relationships between methods and *read* or *write* relationships between methods and attributes. Such information is not found at the UML class diagrams level, but will be available from sequence- and/or state diagrams, provided these are set up at the appropriate level of detail. Alternatively, it would be relatively easy to extract this information directly from the code. The design and implementation of software to do this extraction, whether directly from code or from representations of sequence- or state diagrams, is not considered further in this thesis. Instead, it is simply assumed that the required information is available.

This category of information is required because of the following two points:

a. It is needed to check preconditions of some refactorings. Refactoring precondition, as will be explained later, is important to ensure the behaviour preservation of the refactoring. For example, one of the precondition conjuncts of the primitive refactoring **deleteMethod** that is used to delete a specific method *Methn* from the class diagram is: "The method *Methn* should not be referenced (*called*) by any other *object* elements in the entire system". The information extracted from the class diagram alone is not sufficient to check such a condition. If the system has two classes A and *B* where a method in class *A calls* another method in class *B,* then, the UML class diagram may reflect an *association* relation between the two classes. However, the class diagram does not indicate the reason for the *association*.

b. Some refactorings involve a restructuring of this extra information without modifying anything in the class diagram itself. For example, a refactoring may be used to redirect direct access to a certain attribute through read and/or write methods instead (getter and/or setter). The class diagram is not affected by this refactoring, but the relations between the different members will be changed. The refactoring tool should keep track of such modifications because they are needed:

   - for future refactorings (to check preconditions, for example); or

   - to modify the code or *other* UML diagrams, such as state and sequence diagrams.

Note that the latter point suggests a theme that will not be pursued further in this thesis, namely the notion of keeping various representations of the system consistent with one another, where these may be the system's class diagrams, sequence diagrams, state diagrams, its code, etc. The focus of this thesis will remain on refactoring at the class diagram level, with the aforementioned exception related to access information, so as to address a relatively wide range of refactorings.

The set of logic-terms are accordingly classified into two groups, where each group corresponds to one of the two specific kinds of UML vocabularies. The first group is concerned with *object* elements of the UML class diagram. All the facts in this group are extracted directly from the UML class diagram of the system under consideration. Logic-terms of this group are:

- **package** logic-terms
- **class** logic-terms
- **method** logic-terms
- **attribute** logic-terms
- **parameter** logic-terms

The second group of the logic-terms is concerned with *relation* elements of the UML class diagram. Part of these logic-terms are extracted from the class diagram (*extends*, *association* and *type* relations), while the rest are assumed to have been extracted from the code-level implementation of the system (*read*, *write* and *call*). Logic-terms of this group are:

- **extends** logic-terms
- **association** logic-terms
- **read** logic-terms
- **write** logic-terms
- **call** logic-terms
- **type** logic-terms

The logic-terms of the system under consideration are represented as Prolog facts in the proposed refactoring tool. In the rest of the thesis, the concepts logic-term and Prolog fact will be regarded as exchangeable and have the same meaning.

In general, the first argument of each logic-term (fact) is a unique identifier for the model element (*object* or *relation*). The other arguments are properties of that element (name, definition type and access mode) or are foreign identities for other model elements. In the following two sections (3.2 and 3.3), each group of logic-terms will be presented in detail.

3.2 Object Element Logic-Terms

This group of logic-terms includes all the logic-terms that are used to represent the *object* elements of the UML class diagrams. Logic-terms of this group are:

A. package(*PID, OwnerID, PName, CsList* **)** is used to represent *package* object elements of the UML class diagram. The description of arguments of the package logic-term is as follows:

- *PID* is the unique identifier of the package.
- *OwnerID* is the unique identifier of the container where the package is identified.
- *PName* is the name of the package.
- *CsList* is a list that contains the unique identifiers of all the classes defined in the package.

B. class(*CID, PID, CName, AccMode, MethsList, AttrsList* **)** is used to represent *class* object elements of the UML class diagram. The description of arguments of the class logic-term is as follows:

- *CID* is the unique identifier of the class.
- *PID* is the unique identifier of the package in which the class resides.
- *CName* is the name of the class.
- *AccMode* is the access mode of the class.
- *MethsList* is a list that contains the unique identifiers of all the methods defined in the class.
- *AttrsList* is a list that contains the unique identifiers of all the attributes defined in the class.

C. attribute(*AttrID, CID, AttrName, DefType, AccMode* **)** is used to represent *attribute* object elements of the UML class diagram. The description of arguments of the attribute logic-term is as follows:

- *AttrID* is the unique identifier of the attribute.
- *CID* is the unique identifier of the class where the attribute is identified.
- *AttrName* is the name of the attribute.
- *DefType* is the definition type of the attribute.

- *AccMode* is the access mode of the attribute.

Note: In the rest of the thesis, a distinction between two different definition types is made (*basic* and *complex* definition types) as follows:

a. Basic type: used when the definition type is basic (*int, float, etc*). It takes the following format:

type(*basic, Tname, Num*), *Num* >= 0. (Zero if the variable is not array)

- *basic* stands for basic types like *int*, *float*, *double*, *etc*.
- *Tname* is the type name (*int, float, etc*).
- *Num* stands for the dimension of an array. Zero is used for simple types (i.e. not array).

b. Complex type: used when the definition type is complex (*class*, *interface, etc*). It takes the following format:

type(*complex, ObjectID/ObjectName, Num*), *Num* >= 0.

- *complex* stands for complex types like *class* or *interface*.
- *ObjectID* is the unique identifier (ID) of that object (*class*, *interface, etc*). For example, if the definition type of an attribute *Attn* is a class *A* then this argument will be the ID of *A*. When the user specifies the definition type of an object then the user just enter the name of the object *ObjectName*. The tool then takes the responsibility of storing the ID of that object.
- *Num* stands for the dimension of an array. Zero is used for simple types (i.e. not array).

D. method(*MethID, CID, MethName, RetType, AccMode, PrmsList*) is used to represent *method* object elements of the UML class diagram. The description of arguments of the method logic-term is as follows:

- *MethID* is the unique identifier of the method.
- *CID* is the unique identifier of the class where the method is identified.
- *MethName* is the name of the method.
- *RetType* is the definition type of the return value of the method.
- *AccMode* is the access mode of the method.
- *PrmsList* is a list that contains the unique identifiers of all the parameters defined in the method. The order of these IDs represents the order of the parameters in the method.

E. parameter(*PrmID, MethID, PrmName, DefType*) is used to represent *parameter* object elements defined in methods of the UML class diagram. The description of arguments of the parameter logic-term is as follows:

- *PrmID* is the unique identifier of the parameter.
- *MethID* is the unique identifier of the method where the parameter is identified.
- *PrmName* is the name of the parameter.
- *DefType* is the definition type of the parameter.

3.3 Relation Element Logic-Terms

This group of logic-terms includes all the logic-terms that are used to represent the *relation* elements of the UML class diagrams. Each *relation* logic-term represents a specific *relation* that may exist between two object elements in the UML class diagram. All the *relation* logic-terms have the same arguments as the following:

RelationType(*RID, Label, SourceID, DestinationID*)

Where

- *RID* is the unique identifier of the *relation*.
- *Label* is the label of the *relation*.
- *SourceID* is the unique identifier of the source *object* element of the *relation*.
- *DestinationID* is the unique identifier of the destination *object* element of the *relation*.

Logic-terms of this group are the following:

A. extends(*RID, Label, SourceID, DestinationID*) is used to represent an *extends* (generalization, specialization) relation that may exist between two *object* elements. For example, it may be used to represent the relation between two classes *A* and *B* where the first class *A* (with unique identifier *SourceID*) is the superclass of the second class *B* (with unique identifier *DestinationID*).

B. association(*RID, Label, SourceID, DestinationID*) is used to represent an *association* relation that may exist between two *object* elements. For example, it may be used to represent the relation between two classes *A* and *B* where the first class *A* (with unique identifier

SourceID) is the source of the relation and the second class *B* (with unique identifier *DestinationID*) is the destination of the relation.

C. read(*RID, _, SourceID, DestinationID* **)** is used to represent a *read* relation that may exist between two *object* elements. For example, it may be used to represent the relation between a method *Methn* and an attribute *Attn* where at the code-level one or more statements in the method *Methn* access the attribute *Attn* in a *read* mode. The method *Methn* (with unique identifier *SourceID*) is the source of the relation and the attribute *Attn* (with unique identifier *DestinationID*) is the destination of the relation.

D. write(*RID, _, SourceID, DestinationID* **)** is used to represent a *write* relation that may exist between two *object* elements. For example, it may be used to represent the relation between a method *Methn* and an attribute *Attn* where at the code-level one or more statements in the method *Methn* access the attribute *Attn* in a *write* mode. The method *Methn* (with unique identifier *SourceID*) is the source of the relation and the attribute *Attn* (with unique identifier *DestinationID*) is the destination of the relation.

E. call(*RID, _, SourceID, DestinationID* **)** is used to represent a *call* relation that may exist between two *object* elements. For example, it may be used to represent the relation between two methods *MethX* and *MethY* where at the code-level one or more statements in the method *MethX* call the method *MethY*. The method *MethX* (with unique identifier *SourceID*) is the source of the relation and the method *MethY* (with unique identifier *DestinationID*) is the destination of the relation.

F. type(*RID, _, SourceID, DestinationID* **)** is used to represent a *type* relation that may exist between two *object* elements. For example, it may be used to represent the relation between an attribute *Attn* and a class *C* where the definition type of the *Attn* is class *C*. The attribute *Attn* (with unique identifier *SourceID*) is the source of the relation and the class *C* (with unique identifier *DestinationID*) is the destination of the relation.

Note 1: The second argument *Label* in the logic-terms *read*, *write*, *call* and *type* is ignored. This is because the *read*, *write* and *call* relations do not appear in the original UML class diagram, and they just added to the logic representation of the system as extra information for refactoring purposes. In the case of the *type* relation, it simply does not have a label in the UML class diagram.

Note 2: At the code-level, if a method *Methn* accesses (*reads*) an attribute *Attn* more than once, then at the logic-based representation level these *reads* will be represented by just one *read* relation from *Methn* to *Attn*. The same apply for the *write* and *call* relations.

3.4 Example

Figure 3.1(a) shows a UML class diagram for a simple system *SimpleSys*. The system has a package *D* with two classes *B* and *C* defined in the package.

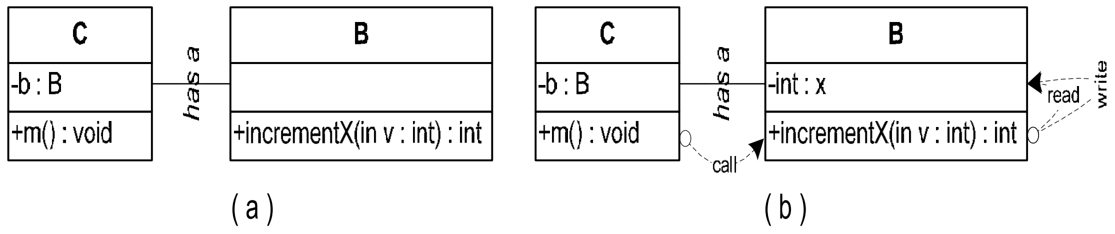


Figure 3.1: A simple UML class diagram of the *SimpleSys*

```
 package D;
 public class C {
   private B b;
   public void m(){
     System.out.println(b.incrementX(10));}
 }  //end of class C
class B{
     private int x;
     public int incrementX(int v){
     x=x+v;
     return x;}
} //end of class B
```

Figure 3.2: A code-level implementation of the *SimpleSys*

As mentioned in section 3.1, *access-related* information that describes the references between the different *object* elements in the UML class diagram is needed for refactoring. This information is extracted from the code-level of the system. For clarity, such information is represented as dashed arrows in Figure 3.1(b). Figure 3.2 shows the code-level implementation of the system from which this information is extracted. For simplicity, the main method in class *C* and the constructors in the different classes are omitted from the code.

In the following, a detailed explanation is given of where each one of the dashed arrows in Figure 3.1(b) is extracted:

- The *write* relation from the method *B.incrementX* to the attribute *B.x* (shown in Figure 3.1(b)) is extracted from line 10 of the code. The value of the attribute *B.x* is updated by the left side of the assignment statement $\underline{\boldsymbol{x}}= x + v$. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the attribute *B.x* will be referenced (*updated*) by the code implemented in the method *B.incrementX*.

- The *read* relation from the method *B.incrementX* to the attribute *B.x* is extracted from line 10 of the code. The value of the attribute *B.x* is read by the right side of the assignment statement $x= \underline{\boldsymbol{x}} + v$. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the attribute *B.x* will be referenced (*read*) by the code implemented in the method *B.incrementX*.

- The *call* relation from the method *C.m* to the method *B.incrementX* is extracted from line 5 of the code. The method *B.incrementX* is called by the statement *b.incrementX(10)* which is implemented in the method *C.m*. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the method *B.incrementX* will be referenced (*called*) by the code implemented in the method *C.m*.

Figure 3.3 shows the list of logic-terms (*Prolog facts*) for the UML class diagram in Figure 3.1. For example, in the fact

class(2, 0, C, public, [2001], [20001]).

- The first argument represents the unique identifier of class C.
- The second argument represents the unique identifier of the container of class C, which is the package with unique identifier 0.
- The third argument is the name of the class.
- The forth argument is the access mode of the class C.
- The fifth argument is a list that contains the unique identifiers of all the methods defined in the class C. In this case, it is just one method with unique identifier 3001.
- The last argument is a list that contains the unique identifiers of all the attributes defined in the class C. In this case, it is just one attribute with unique identifier 30001.

```
package(0, 00, D, [1, 2]).
class(1, 0, B, default, [1001], [10001]).
method(1001, 1, incrementX, type(basic,int,0),  public, [100001]).
attribute(10001, 1, x, type(basic, int, 0), private).
parameter(100001, 1001, v, type(basic, int, 0)).
read(1000001, _, 1001, 10001).
write(1000002, _, 1001, 10001).
class(2, 0, C, public, [2001], [20001]).
method(2001, 2, m, type(basic, void, 0), public, []).
attribute(20001, 2, b, type(complex, 1, 0), private).
type(2000001, _, 20001, 1).
call(2000002, _, 2001, 1001).
association(2000003, has a, 2, 1).
```

Figure 3.3: Underlying logic representations of the *SimpleSys*

3.5 Reflection on this Chapter

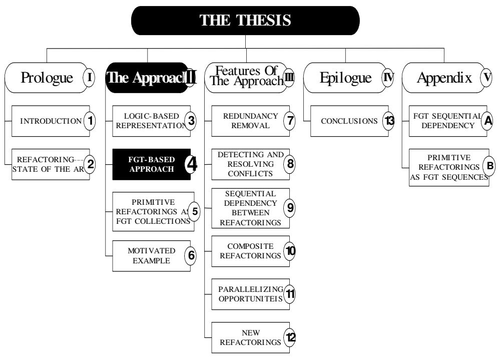
There foregoing schema is used in the current thesis as a knowledge base to represent a UML-specified system that is manipulated by a prototype Prolog refactoring tool to refactor the system according to user-specified refactorings. The tool, therefore, contains Prolog rules to apply these refactorings. It also requires rules to check that preconditions of refactorings are satisfied before their application can be attempted. Of course, in the present thesis, all of the refactorings happen in terms of FGTs, and the tool has been designed to operate precisely at this FGT-level. The forthcoming chapters will elaborate further on these themes.

However, it might be noted in passing that the schema given above could also be used as a basis for issuing queries about a UML system—for example, for finding all classes that have a certain characteristic in the system. While this theme will not be further explored in this thesis, it appears to be a peripheral contribution of the thesis that could conceivably be exploited in developing such a Prolog-based application.

CHAPTER 4

FGT-BASED APPROACH

4.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT SEQUENCES B

The main focus of this chapter is to give detailed explanations and descriptions of the set of FGTs to represent and construct any refactorings in the proposed refactoring tool. This chapter is the ground base for the remaining chapters. Each of those later chapters may be read independently, provided that the reader is familiar with the contents of this chapter, the rest of which is organized as follows.

In section 4.2 the concept FGT is described, and details the two types of FGTs (*Object Element* and *Relational Element* FGTs) are given. Full details about the format, implementation, and the set of precondition conjuncts for each FGT are given.

In section 4.3 an algorithm is introduced that is used to allocate the collection of the FGTs that are related to one refactoring in a data structure called an FGT Directed Acyclic Graph (FGT-DAG). Since the algorithm accounts for the sequential dependencies that may occur between the different FGTs in the refactoring, the section also provides a detailed explanation of the sequential dependencies between the different FGTs.

In section 4.4 the relationship between the set of FGTs and primitive—as well as composite—refactorings is discussed. The section describes a vision in which the proposed set of FGTs constitutes the core of the refactoring system, and suggests new terminology for describing refactoring precondition.

4.2 Fine-Grain Transformations (FGTs)

An FGT is an *abstract operation* on a UML model—i.e. a UML model will always be one of the implicit operands of an FGT, and this model will always undergo an incremental *atomic* change as a result of applying an FGT to it. The change can be regarded as atomic in the sense that it cannot be broken down into further smaller change steps from the modeling perspective. The operation is abstract in the sense that it could be specified in a wide variety of concrete syntactic representations.

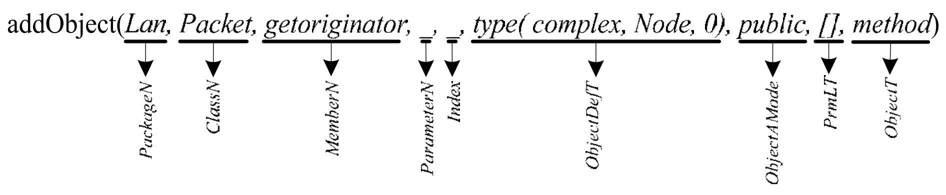
Throughout this thesis, a concrete syntax that resembles Prolog predicates will be used to specify FGTs. This choice of concrete syntax was made to support the Prolog prototype refactoring tool that has been built to illustrate the various ideas. As described in the previous chapter, the UML class diagram is itself stored as a set of facts in the Prolog database. As will be seen below, the concrete syntax of each FGT has to uniquely identify the various components of the UML class diagram that are to change, and it also has to indicate the nature of the change. In general, the nature of the change is encapsulated in the name of the FGT, and the UML components that are affected are specified as arguments of it.

The set of FGTs that have been identified are closely related to the vocabulary and semantics of standard UML mentioned in the previous chapter and they are accordingly classified into the two groups used in chapter 3.

The first group is concerned with all the transformation operations whose characterising operands are *object* elements of the UML class diagram. In the rest of the thesis, these FGTs are called *Object Element FGTs.* FGTs of this group are:

- **addObject** FGT: used to add *object* elements to the class diagram.
- **renameObject** FGT: used to change the name of an *object* element.
- **changeOAMode** FGT: used to change the access mode of an *object* element.
- **changeODefType** FGT: used to change the definition type of an *object* element.
- **deleteObject**: used to delete *object* element from the class diagram.

As an example of FGTs in this group, the following FGT is used to add to the class diagram an *object* element with name *getoriginator* and access mode *public.* It is to be added to the class *Packet* that is in the package *Lan*. The *object* will return one value of type *Node* class. The last argument of the FGT tells the tool that the added *object* in this FGT is of type *method*. The empty list *PrmLT* indicates that the added method will have no parameters.



```
addObject(Lan, Packet, getoriginator, _, _, type( complex, Node, 0), public, [], method)
```

After applying this FGT, the following fact will be added to the underlying database of facts that represents the class diagram of the system under consideration:

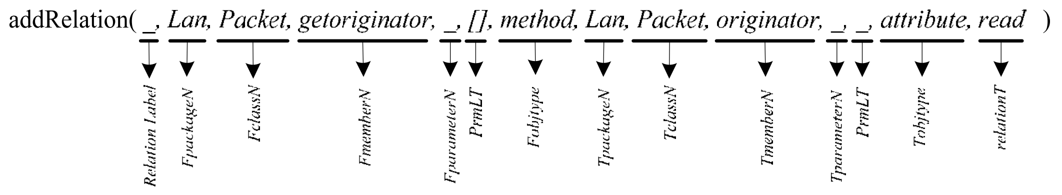
method(46, 2, getoriginator, type(complex,1, 0), public, []).

Note that from the information presented in section 3.2, number 46 will be the unique identifier of the new method. Number 2 is the unique identifier of the class *Packet* where the new method will be defined. Number 1 in the term *type* is the unique identifier of the definition type of the return value of the method, which is in this case the class *Node*.

The second group of FGTs is concerned with all the transformation operations that work on *relational* elements of a UML class diagram. These FGTs will be called *Relational Element FGTs.* FGTs of this group are:

- **addRelation** FGT: used to add a *relational* element between two *object* elements.
- **renameRelation** FGT: used to change the label of a *relational* element.
- **deleteRelation** FGT: used to delete a *relational* element that exists between two *object* elements.

As an example of FGTs in this group, the following FGT is used to add a *read* relation from the method *Lan.Packet.getoriginator* to the attribute *Lan.Packet.originator*.



After applying this FGT, the following fact will be added to the underlying database facts that represent the class diagram of the system under consideration:

read(47, _, 46, 2002).

Number 47 will be the unique identifier of the new *read* relation. Number 46 is the unique identifier of the source *object* of the relation, which is *Lan.Packet.getoriginator* method. Number 2002 is the unique identifier of the destination *object* of the relation, which is *Lan.Packet.originator* attribute. As mentioned in section 3.3, the label for the *read*, *write*, *call* and *type* relations is omitted.

Each FGT of the two groups has a set of precondition conjuncts (i.e. X and Y and Z and …) that need to be satisfied by the system in order to consider it as a legal transformation operation. In some cases, one or more of these conjuncts is itself a number of disjuncts (i.e. (X or Y)). A procedure called **FGTPrecondConj**(*FGT*) is implemented in the refactoring tool for each one of the proposed FGTs. FGTs precondition conjuncts will play an important role in preserving the behaviour of the system at the time of refactoring, as will be shown in section 4.4. For example, in order to apply the FGT:

addObject(*Lan, Packet, getoriginator, _, _, type(complex, Node, 0), public, [], method*)

The underlying system should have a class with name *Packet* in the package *Lan*; and this class should not contain a method *getoriginator* with empty parameter list. The method *getoriginator* should also not be inherited from any of the ancestors of class *Lan.Packet*. In addition, the return definition type of the method should be valid and accessible. The access mode of the created method should also be valid. The precondition conjuncts for this FGT, as implemented in the prototype tool, are specified as follows:

FGTPrecondConj(addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method*) **):-**

existsObject(*Pn, Cn, class*),
not(existsObject(*Pn, Cn, Methn, PrmLT, method*)),
not(isInherited(*Pn, Cn, Methn, PrmLT, method*)),
validDefType(*ODefT*),
canAccessType(*ODefT*),
validOAMode(*OAMode, method*).

Note that the comma (,) between the two conjuncts retains the Prolog semantics of a "*logical and*" between two rules. As another example, in order to apply the FGT

addRelation(_,*Lan, Packet, getoriginator, _, [], method, Lan, Packet, originator, _, _, attribute, read*)

The underlying system should have the method *Lan.Packet.getoriginator* and the attribute *Lan.Packet.originator.* The system may not already have a *read* access between the method *Lan.Packet.getoriginator* and the attribute *Lan.Packet.originator*. In addition, the location of the source *object Lan.Packet.getoriginator* and the destination *object Lan.Packet.originator* in the model together with the access mode of the destination *object Lan.Packet.originator* play an important role in determining the applicability of the previous addRelation FGT. The precondition conjuncts for this FGT, as implemented in the prototype tool, are specified as follows:

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn, _,FPrmLT, method, TPn, TCn, TAttn, _, _, attribute, RelT*))**:-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),
existsObject(*TPn, TCn, TAttn, attribute*),
not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TAttn, attribute,RelT*)) ,
[(objectAMode(*TPn, TCn, TAttn, attribute, private*), *FPn.FCn=TPn.TCn*) |
(objectAMode(*TPn, TCn, TAttn, attribute, default*), *FPn=TPn*) |
(objectAMode(*TPn, TCn, TAttn, attribute, protected*), (subClass(*FPn,FCn, TPn, TCn*) |
FPn=TPn)) | objectAMode(*TPn, TCn, TAttn, attribute, public*)]**.**

Note that the comma (|) between the two conjuncts retains the Prolog semantics of a "*logical or*" between two rules.

A detailed explanation of the addRelation's precondition conjuncts, as well as those of all the other FGTs, will be discussed later in this section. In the following two subsections 4.2.1 and 4.2.2, each group of FGTs together with their set of precondition conjuncts will be presented in detail.

The presentation of each FGT will be in the following style. Firstly, the format of the Prolog term used to represent that FGT in the system is explained. The explanation includes an explanation of each of the term's arguments. Then the Prolog rule used to check the preconditions of the FGT is given. If F represent the Prolog term of some FGT, then this precondition rule has the general form:

FGTPrecondConj(F) :- C1, C2, … Cn.

where C1 … Cn are Prolog terms (containing arguments suitably derived from the arguments of F) representing the n precondition conjuncts that need to be checked against the existing system description. English narrative is given alongside these terms to explain what their meaning is. The Prolog rules used to check the truth-value of the terms C1, … Cn are not discussed here, but are fairly straightforward. Similarly, the postconditions resulting from applying each FGT to the system under consideration are not explicitly stated, but may easily be inferred from the nature of the FGT: the relevant object or relation has been added / renamed / deleted; or the access mode or definition type of an object element has been changed. In predicate logic, these could typically be represented by formulae which assert the existence of some object or relation that previously did not exist, and / or the non-existence of some object or relation that previously existed. In Prolog, these postconditions are operationally realised by the insertion into, or deletion from the Prolog database of relevant logical terms (as discussed in Chapter 3) representing these objects and relations.

In terms of the classical notation for total correctness proposed by Hoare, the axiomatic semantics of FGT F whose precondition is C1 ^ … ^ Cn, and whose postcondition is P1 ^ … ^ Pm could be given as:

{C1 ^ C2 ... ^ Cn} F {P1 ^ P2 … ^ Pm}

i.e. if C1 and C2 and … Cn are true (of the system under consideration) before applying F to it, and F is applied to this system, then F will terminate and P1 and P2 and … Pm will be true. Although the axiomatic semantics of the various FGTs are not explicitly provided below, they are all easily derivable from the information given.

The question may be asked: is the definition of each FGT sound in the sense that its underlying axiomatic semantics correctly specifies what is intended? For example, have all the precondition conjuncts C1, C2 … Cn been correctly identified to add / delete / rename the relevant object or relation according to the rules of the language in question (in the present case, UML representing an underlying Java system)? A formal proof of this kind of soundness is beyond the scope of this thesis. Under the circumstances, the best that could be done was to manually check the soundness of each FGT. While this does not, of course, guarantee soundness, it is hoped that the explicit provision of preconditions given below will allow others to scrutinise the axiomatic semantics for the type of soundness mentioned above.

Similarly, the question may be asked: is the class of FGTs provided in the forthcoming sections complete in the sense that no other possible FGTs can be defined? Again, there does not seem to be any easy way of formally guaranteeing this. Later in this chapter, it will be seen, however, that the class of FGTs defined is sufficient for building all commonly known primitive refactorings. In this sense, the class of FGTs defined below can be said to be complete.

4.2.1 Object Element FGTs

This group of FGTs includes all FGTs that are used to manipulate *object* elements of a UML class diagram. (Recall that *object* elements in the simplified UML meta-model include classes, methods, attributes and parameters.) By using these FGTs, the developer can add, rename, change access mode, change definition type, or delete *object* elements from a UML class diagram.

4.2.1.1 addObject FGT

The addObject FGT is used to add *object* elements to the UML class diagram. It is used to add class, method, attribute, parameter *object* elements to the class diagram. In general, it takes the following format:

addObject(*Pn, Cn, Memn, Prmn, Index, ODefT, OAMode, PrmLT, OT*)

where

- *Pn* is the name of the package. It is used when the *object* to be added is a package, class, method, attribute or parameter.
- *Cn* is the name of the class. It is used when the *object* to be added is a class, method, attribute or parameter.
- *Memn* is the name of the member (method or attribute). It is used when the *object* to be added is a method, attribute or parameter.
- *Prmn* is the name of the method's parameter. It is used when the *object* to be added is a parameter.
- *Index* is the index (order) of the parameter in the method's parameter list. It is used when the *object* to be added is a parameter.
- *ODefT* is the *object* definition type. It is used when the *object* to be added is a method, attribute or parameter. If the *object* is a method, then *ODefT* refers to the return type of that method. If the *object* is an attribute or parameter, then *ODefT* refers to the definition type of the attribute or parameter.
- *OAMode* is the access mode of the *object* (*public, protected, default, private*).
- *PrmLT* is the list of method's parameters. It is used when the *object* to be added is a method or parameter. If it is a method then the list *PrmLT* will contain all the parameters that are declared in the method. These parameters will be ordered in the list according to their definition order in the method arguments. Each element in the list is represented by the pair (*Prmn, PrmDefT*) where *Prmn* is the name of the parameter and *PrmDefT* is the definition

type of that parameter. If the *object* to be added is a parameter then the list *PrmLT* will contain the definition type of all parameters that are declared in the method. These definition types will be ordered in the list according to the order of their associated parameters in the method arguments. In this case, the list *PrmLT* is used to specify the signature of the method. For the rest of the thesis, a method's signature is specified by the method name together with its associated *PrmLT*.

- *OT* is the type of the *object* (*class, method, attribute,* or *parameter*).

The set of arguments and precondition conjuncts that are used for the FGT addObject are dependent on the type of *object* element that is to be added to the UML class diagram using that FGT, as shown below:

A. addObject(*Pn, Cn, _, _, _, _, OAMode, _, class***)**

As indicated in the last argument, this FGT is used to add a new class *Cn* in the package *Pn* with access mode *OAMode*. The new class will be empty and standalone. Empty means that, it has no members (attributes or methods). Standalone means, that it has no superclass or subclasses. All the members and super- or subclass *relations* will be added to the new class at a later stage.

To apply this FGT on the underlying system the following should hold.

- The package *Pn* should be already declared in the system.
- The class name (*Cn*) should be distinct from those all classes declared in the package *Pn*.
- The access mode *OAMode* should be a valid access mode.

FGTPrecondConj(addObject(*Pn, Cn, _, _, _, _, OAMode, _, class*) **):-**

existsObject(*Pn, package*), not(existsObject(*Pn, Cn, class*)), validOAMode(*OAMode, class*).	1. *Package Pn declared in the system.* 2. *Class Pn.Cn not declared in the system.* 3. *The access mode OAMode is a valid access mode for classes.*

B. addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method***)**

As indicated in the last argument, this FGT is used to add a new method *Methn* with a parameter list *PrmLT* in the class *Pn.Cn*. The new method will have an access mode *OAMode* and a return type defined by the argument *ODefT*.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The signature of the method should be distinct from those all methods declared in the class *Pn.Cn* or any of its ancestor classes.
- The access mode *OAMode* and the definition type of the return value *ODefT* should be valid.
- The type of the return value *ODefT* should be accessible.

Note that the second precondition conjunct means that the method should not be inherited by the class *Pn.Cn* from one of its ancestors. This condition is used to avoid redefining inherited members. Adding a member *x* in a class *A* while it is defined in *A's* ancestors will redefine the member *x* in the class *A* and all descendants of *A* because they will use the new version of *x*, and this will therefore change the behaviour of the system. On the other hand, adding a member *x* in a class *A* while it is defined in *A's* descendants will not change the definition of *x* in *A's* descendant classes. The behaviour of the system will therefore not change.

The last precondition conjunct is important when the type of the return value is complex (not basic). For example, if the return value is of type class, then the access mode of that class should be accessible.

FGTPrecondConj(addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method*))**:-**

existsObject(*Pn, Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(existsObject(*Pn, Cn, Methn, PrmLT, method*)),	2. *Method Pn.Cn.Methn with PrmLT not declared in the system.*
not(isInherited(*Pn, Cn, Methn, PrmLT, method*)),	3. *Method Methn with PrmLT not declared in any Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	4. *The return definition type of the method is valid.*
validOAMode(*OAMode, method*).	5. *The access mode OAMode is valid.*
canAccessType(*ODefT*)	6. *The return definition type of the method is accessible.*

C. addObject(*Pn, Cn, Attn, _, _, ODefT, OAMode, _, attribute***)**

This FGT, as indicated above, is used to add a new attribute *Attn* in the class *Pn.Cn* with access mode *OAMode*. The type of the new attribute is defined by the argument *ODefT*.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The attribute name *Attn* should be distinct from those all attributes declared in the class *Pn.Cn* or any of its ancestor classes.
- The access mode *OAMode* and the definition type ODefT should be valid.
- The definition *type ODefT* should be accessible.

FGTPrecondConj(addObject(*Pn, Cn, Attn, _, _, ODefT, OAMode,_, attribute*) **):-**

existsObject(*Pn, Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(existsObject(*Pn, Cn, Attn, attribute*)),	2. *Attribute Pn.Cn.Attn not declared in the system.*
not(isInherited(*Pn, Cn, Attn, attribute*)),	3. *Attribute Attn not declared in any of Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	4. *The definition type ODefT is valid.*
validOAMode(*OAMode, attribute*),	5. *The access mode OAMode is valid.*
canAccessType(*ODefT*)).	6. *The definition type ODefT is accessible.*

D. addObject(*Pn, Cn, Methn, Prmn, Index, ODefT, _, PrmLT, parameter***)**

This FGT as indicated from the last argument is used to declare a new parameter *Prmn* in the method *Pn.Cn.Methn* with *PrmLT*. The type of the new parameter is defined by the argument *ODefT*. The new parameter will be added at the index *Index* of the list of the method parameters. If *Index* is occupied by another parameter *x*, then parameter *x* and all the subsequent parameters will be shifted one-step to the right.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system.
- The parameter *Prmn* may not already be declared in the method *Pn.Cn.Methn* with *PrmLT*.
- After adding the parameter *Prmn* to the list of parameters of the method *Methn* with *ParmLT,* the method *Methn* with the modified parameters list *ParmALT* should not be declared in the class *Pn.Cn* or any of its ancestor classes.

- The definition type *ODefT* should be valid and accessible.

FGTPrecondConj(addObject(*Pn, Cn, Methn, Prmn, Index, ODefT, _, PrmLT, parameter*) **):-**

existsObject(*Pn, Cn, Methn, PrmLT, method*),	1. *The method Pn.Cn.Methn with PrmLT declared in the system.*
not(existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*)),	2. *The parameter Prmn not declared in the method Pn.Cn.Methn with PrmLT.*
not(existsObject(*Pn, Cn, Methn, ParmALT, method*)),	3. *The method Pn.Cn.Methn with ParmALT should not be declared in the class Pn.Cn , where ParmALT is the result parameter list type for the method Pn.Cn.Methn after adding Prmn to it.*
not(isInherited(*Pn, Cn, Methn, PrmALT, method*)),	4. *Method Methn with PrmALT not declared in any of Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	5. *The definition type ODefT is valid.*
canAccessType(*ODefT*)).	6. *The definition type ODefT is accessible.*

4.2.1.2 renameObject FGT

This FGT is used to change the name of one of the *object* elements that is already declared in the class diagram. It takes the following format:

renameObject(*Pn, Cn, Memn, Prmn, PrmLT, OT, ONewN*)

Where:

- *ONewN* is the new name of the *object*.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT renameObject are dependent on the type of *object* element that is to be renamed, as shown below:

A. renameObject(*Pn, Cn, _, _, _, class, ONewN***)**

This FGT is used to change the name of the class *Pn.Cn* to another name *Pn.ONewN*.

To apply this FGT on the underlying system, the class *Pn.Cn* should be already declared in the system and the name *ONewN* should be distinct from those all classes declared in the package *Pn*.

FGTPrecondConj(renameObject(*Pn, Cn, _, _, _, class, ONewN*) **):-**

existsObject(*Pn, Cn, class*), not(existsObject(*Pn, ONewN, class*)).	1. *Class Pn.Cn declared in the system.* 2. *Class Pn.ONewN not declared in the system.*

B. renameObject(*Pn, Cn, Methn,_, PrmLT, method, ONewN***)**

This FGT is used to change the name of the method *Pn.Cn.Methn* with *PrmLT* to another name *Pn.Cn.ONewN***.**

To apply this FGT on the underlying system, the method *Pn.Cn.Methn* should be already declared in the system and the name *ONewN* should be distinct from those all methods with *PrmLT* that are declared in the class *Pn.Cn* or any of its ancestor classes.

FGTPrecondConj(renameObject(*Pn, Cn, Methn, _, PrmLT, method, ONewN*) **):-**

existsObject(*Pn, Cn, Methn, PrmLT, method*), not(existsObject(*Pn, Cn, ONewN, PrmLT, method*)), not(isInherited(*Pn, Cn, Methn, PrmLT, method*)).	1. *Method Pn.Cn.Methn with PrmLT declared in the system.* 2. *Method Pn.Cn.ONewN with PrmLT not declared in the system.* 3. *Method Methn with PrmLT not declared in any of the Pn.Cn's ancestor of classes.*

C. renameObject(*Pn, Cn, Attn, _, _, attribute, ONewN***)**

This FGT is used to change the name of the attribute *Pn.Cn.Attn* to another name *Pn.Cn.ONewN***.** To apply this FGT on the underlying system, the attribute *Pn.Cn.Attn* should be already declared in the system and the name *ONewN* should be distinct from those all attributes that are declared in the class *Pn.Cn* or any of its ancestor classes.

FGTPrecondConj(renameObject(*Pn, Cn, Attn, _, _, attribute, ONewN*) **):-**

existsObject(*Pn, Cn, Attn, attribute*), not(existsObject(*Pn, Cn, ONewN, attribute*)),	1. *Attribute Pn.Cn.Attn declared in the system.* 2. *The attribute Pn.Cn.ONewN not declared in the system.*

not(isInherited(*Pn, Cn, Attn, attribute*)).	3. *Attribute Attn not declared in any of the Pn.Cn's ancestor of classes.*

D. renameObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter, ONewN*)

This FGT is used to change the name of the parameter *Prmn* that is declared in the method *Pn.Cn.Methn* with *PrmLT* to another name *Pn.Cn.Methn.ONewN***.**

To apply this FGT on the underlying system, the method *Pn.Cn.Methn* should be already declared in the system and the name *ONewN* should be distinct from those all parameters that are declared in the method *Pn.Cn.Methn* with *PrmLT*.

FGTPrecondConj(renameObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter, ONewN*))**:-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*),	1. *Parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.*
not(existsObject(*Pn, Cn, Methn, ONewN, PrmLT, parameter*)).	2. *The parameter ONewN not declared in the method Pn.Cn.Methn with PrmLT.*

4.2.1.3 changeOAMode FGT

This FGT is used to change the access mode (*public, protected, default* and *private*) of class, method, or attribute *object* elements from one mode to another. It cannot be applied to the parameter *object* elements, because there is no access mode for these elements in the class diagram. The FGT takes the following format:

changeOAMode(*Pn, Cn, Memn, Prmn, PrmLT, OT, OOldAM, ONewAM*)

Where:

- *OOldAM* is the old access mode of the *object* element.
- *ONewAM* is the new access mode of the *object* element.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

Changing the access mode of *object* X from a higher restricted access mode to a lower one can be done easily without any difficulties because none the references from the other *objects* to the *object* X will be affected. However, changing the access mode of *object* X from a lower restricted access mode to a higher one requires more attention. This is because, if *object* X is referenced by an *object* Y and this reference is allowed only when the access mode of *object* X is that lower restricted one, then changing it to a more restricted one will not allow such a reference from *object* Y to *object* X. To compare the restriction levels of two access modes, the procedure **moreRestLevel**(*OAModex, OAModey*) is used. The procedure returns true if the access mode *OAModex* is more restricted than the access mode *OAModey* and returns false for other cases.

The set of arguments and precondition conjuncts that are used for the FGT changeOAMode are dependent on the type of *object* element that is to be changed, as shown below:

A. changeOAMode(*Pn, Cn, _, _, _, class, OOldAM, ONewAM***)**

This FGT is used to change the class access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the class can be *public* or *default*. Changing the access mode of the class from a higher restricted access mode (*default*) to a lower restricted one (*public*) can be done without any difficulties because none of the references to the class *Pn.Cn* will be affected. However, changing the access mode of the class from a lower restricted access mode (*public*) to a higher restricted one (*default)* requires more attention. This is because if the class *Pn.Cn* is referenced by any other *object* that is located outside the package *Pn*, after changing the access mode to *default* that reference will not be allowed. Thus, changing the access mode of the class from *public* to *default* requires that the class *Pn.Cn* should not be referenced by any other *object* locate outside the package *Pn*. To verify this, the procedure **referenceOutPackage**(*Pn, Cn, class*) is used. The procedure indicates whether there is any reference to that class from *objects* locates outside the package.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The old access mode *OOldAM* not equal to the new access mode *ONewAM*.
- If the new access mode *ONewAM* is *default* then the class *Pn.Cn* should not be referenced from outside the package *Pn*.

FGTPrecondConj(changeOAMode(*Pn, Cn, _, _, _, class, OOldAM, ONewAM*))**:-**

existsObject(*Pn, Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(*OOldAM=ONewAM*),	2. *Old access mode not equal to the new access mode.*

[(*ONewAM=default*, not(referenceOutPackage(*Pn, Cn, class*))) |	3. *If the new access mode is default then the class Pn.Cn should not be referenced from outside package Pn.*
ONewAM=public].	4. *If the new access mode is public then the class Pn.Cn can be referenced from anywhere (the condition between the two brackets [] will be true).*

B. changeOAMode(*Pn, Cn, Methn, _, PrmLT, method, OOldAM, ONewAM***)**

This FGT is used to change the method access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the method can be *public*, *protected*, *default or private*.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system.
- The old access mode *OOldAM* not equal to the new access mode *ONewAM*.
- If the new access mode *ONewAM* is more restricted than the old one *OOldAM* then changing the access mode will be done easily without any difficulties. For the other cases, conditions 3 to 5 in the following list of precondition conjuncts are used.

FGTPrecondConj(changeOAMode(*Pn, Cn, Methn,_,PrmLT,method, OOldAM,ONewAM*))**:-**

existsObject(*Pn, Cn, Methn, PrmLT, method*),	1. *Method Pn.Cn.Methn with PrmLT declared in the system.*
not(*OOldAM=ONewAM*),	2. *The old access mode not equal to the new access mode.*
[(*ONewAM=private* , not(referenceOutClass(*Pn, Cn, Methn, PrmLT, method*))) |	3. *If the new access mode is private then the method Pn.Cn.Methn should not be accessed from outside the class Pn.Cn.*
(moreRestLevel(*ONewAM, OOldAM*), *ONewAM=protected* , not(referenceOutPckSub(*Pn, Cn, Methn, PrmLT , method*))) |	4. *If the new access mode is protected and it is more restricted than the old one then the method Pn.Cn.Methn should not be accessed from outside the subclasses of the class Pn.Cn or the package Pn.*

(moreRestLevel(*ONewAM, OOldAM*), *ONewAM=default* , not(referenceOutPackage(*Pn, Cn, Methn, PrmLT*))) | moreRestLevel(*OOldAM*, *ONewAM*)].	5. *If the new access mode is default, and it is more restricted than the old one then the method Pn.Cn.Methn should not be accessed from outside the package Pn.* 6. *For all the other cases in which the new access mode is less restricted than the old one then the condition between the two brackets [] will be true.*

C. changeOAMode(*Pn, Cn, Attn, _, _, attribute, OOldAM, ONewAM***)**

This FGT is used to change the attribute access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the attribute can be *public*, *protected*, *default or private*.

To apply this FGT on the underlying system the following should hold.

- The attribute *Pn.Cn.Attn* should already declare in the system.
- The old access mode *OOldAM* may not be equal to the new access mode *ONewAM*.
- If the new access mo*de ONewAM* is more restricted than the old one *OOldAM,* then changing the access m*ode wi*ll be done easily without any difficulties. For the other cases, conditions 3 to 5 in the following list of precondition conjuncts are used.

FGTPrecondConj(changeOAMode(*Pn, Cn, Attn, _, _, attribute, OOldAM, ONewAM*) **):-**

existsObject(*Pn, Cn, Attn, attribute*),	1. *The attribute Pn.Cn.Attn declared in the system.*
not(*OOldAM=ONewAM*),	2. *The old access mode not equal to the new access mode.*
[(*ONewAM=private*, not(referenceOutClass(*Pn, Cn, Attn, attribute*))) |	3. *If the new access mode is private then the attribute Pn.Cn.Attn should not be referenced from outside the class Pn.Cn.*
(moreRestLevel(*ONewAM, OOldAM*), *ONewAM=protected*, not(referenceOutPckSub(*Pn, Cn, Attn, attribute*))) |	4. *If the new access mode is protected and it is more restricted than the old one then the attribute Pn.Cn. should not be accessed from outside the subclasses of the class Pn, Cn or the package Pn.*

(moreRestLevel(*ONewAM, OOldAM*), *ONewAM=default*, not(referenceOutPackage(*Pn, Cn, Attn, attribute*))) |	5. *If the new access mode is default and it is more restricted than the old one then the attribute Pn.Cn.Attn should not be accessed from outside the package Pn.*
moreRestLevel(*OOldAM*, *ONewAM*)].	6. *For all the other cases in which the new access mode is less restricted than the old one then the condition between the two brackets will be true.*

4.2.1.4 changeODefType FGT

The changeODefType FGT is used to change the definition type of the method, attribute and parameter *object* elements in the class diagram from one definition type to another. It does not apply to class *object* elements because these *object* elements do not have type definitions in the class diagram. For the method *object* elements, the changeODefType FGT changes the definition type of the return value of the method.

It takes the following format:

changeODefType(*Pn, Cn, Memn, Prmn, PrmLT, OT, OOldDT, ONewDT*)

Where:

- *OOldAM* is the old definition type of the *object* element.
- *ONewAM* is the new definition type of the *object* element.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT changeODefType are dependent on the type of *object* element that is to be changed, as shown below:

A. changeODefType(*Pn, Cn, Methn,_, PrmLT, method, OOldDT, ONewDT***)**

This FGT is used to change the definition type of the method's return value from an old definition type *OOldDefT* to a new one O*NewDefT*.

To apply this FGT on the underlying system then the method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system and the old definition type *OOldDefT* is not equal to the new one *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn,Cn,Methn,_,PrmLT, method, OOldDT, ONewDT*))**:-**	
existsObject(*Pn, Cn, Methn, PrmLT, method*),	1. *The method Pn.Cn.Methn with PrmLT declared in the system.*
not(*OOldD T=ONewDT*).	2. *The old return type not equal to the new return type.*

B. changeODefType(*Pn, Cn, Attn, _, _, attribute, OOldDT, ONewDT***)**

This FGT is used to change the definition type of an attribute from an old definition type *OOldDefT* to a new one *ONewDefT*.

To apply this FGT on the underlying system then the attribute *Pn.Cn.Attn* should be already declared in the system and the old definition type *OOldDT* is not equal to the new definition type *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn, Cn, Attn, _, _, attribute, OOldDT, ONewDT*))**:-**	
existsObject(*Pn, Cn, Attn, attribute*),	1. *The attribute Pn.Cn.Attn declared in the system.*
not(*OOldDT=ONewDT*).	2. *The old return type not equal to the new return type.*

C. changeODefType(*Pn, Cn, Methn, Prmn, PrmLT ,parameter, OOldDT, ONewDT***)**

This FGT is used to change the definition type of a parameter from an old definition type *OOldDefT* to a new one *ONewDefT*.

To apply this FGT on the underlying system then the parameter *Prmn* should be declared in the method *Pn.Cn.Methn* with *PrmLT* and the old definition type *OOldDT* is not equal to the new one *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn, Cn, Methn, Prmn, PrmLT , parameter, OOldDT, ONewDT*) **):-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*), not(*OOldDT=ONewDT*).	1. *The parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.* 2. *The old return type not equal to the new return type.*

4.2.1.5 deleteObject FGT

The deleteObject FGT is used to delete unreferenced *object* elements from the UML class diagram. It is used to delete class, method, attribute, parameter *object* elements from the class diagram. It is not allowed to delete any *object* element from the system if that *object* is being referenced by any other *object* in the system.

The deleteObject FGT takes the following format:

deleteObject(*Pn, Cn, Memn, Prmn, PrmLT, OT)*

Note: For the description of the arguments in the FGTs the reader is referred to section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT changeOAMode are dependent on the type of *object* element that to be deleted from the UML class diagram using that FGT, as shown below:

A. deleteObject(*Pn, Cn, _, _, _, class*)

This FGT as indicated from the last argument is used to delete an unreferenced empty class *Cn* from the package *Pn.*

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be declared in the system.
- The class is empty—i.e. it has no members (methods or attributes). If the class to be deleted has members, then these members should first be deleted by using other FGTs. This is important to control and manage the FGTs, as will be explained later.
- The class *Pn.Cn* has neither superclass nor subclasses.

- The class *Pn.Cn* is unreferenced from any other *object.*

FGTPrecondConj(deleteObject(*Pn, Cn, _, _, _, class*) **):-**

existsObject(*Pn,Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(members(*Pn, Cn, class*)),	2. *Class Pn.Cn has no members.*
not(supclass(*Pn,Cn,_ ,_*)),	3. *Class Pn.Cn does not have subclasses.*
not(subclass(*Pn,Cn,_,_*)),	4. *Class Pn.Cn does not have superclass.*
not(isReferenced(*Pn,Cn, class*)).	5. *Class Pn.Cn is unreferenced from any other object.*

B. deleteObject(*Pn, Cn, Methn, _, PrmLT, method***)**

As indicated by the last argument, this FGT is used to delete an unreferenced method *Pn.Cn.Methn* with parameter type list *PrmLT*. Note that if the method is indirectly referenced by instances of one of the *Pn.Cn's* subclasses, then the method has to be regarded as a referenced *object* and may not be deleted.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be declared in the system.
- The method is not referenced (*directly or indirectly*) by any other *object*.

FGTPrecondConj(deleteObject(*Pn, Cn, Methn, _, PrmLT, method*) **):-**

existsObject(*Pn,Cn, Methn, PrmLT , method*),	1. *Method Pn.Cn.Methn with PrmLT declared in the system.*
not(isReferenced(*Pn,Cn, Methn, PrmLT, method*)).	2. *Method Pn,Cn.Methn with PrmLT not referenced (directly or indirectly) by any other object in the system.*

C. deleteObject(*Pn, Cn, Attn, _, _, attribute***)**

This FGT as indicated from the last argument is used to delete an unreferenced attribute *Pn.Cn.Attn.* Note that if the attribute is indirectly referenced by instances of one of the *Pn.Cn's* subclasses, then the attribute has to be regarded as a referenced *object* and may not be deleted.

To apply this FGT on the underlying system the following should hold.

- The attribute *Pn.Cn.Attn* should be declared in the system.
- The attribute is not referenced (*directly or indirectly*) by any other *object*.

FGTPrecondConj(deleteObject(*Pn, Cn, Attn, _, _, attribute*) **):-**

existsObject(*Pn,Cn, Attn, attribute*),	1. *Attribute Pn.Cn. Attn declared in the system.*
not(isReferenced(*Pn,Cn, Attn, attribute*)).	2. *Attribute Pn.Cn.Attn not referenced (directly or indirectly) by any other object in the system.*

D. deleteObject(*Pn, Cn, Memn, Prmn, PrmLT, parameter***)**

This FGT as indicated from the last argument is used to delete parameter *Prmn* from the method *Pn.Cn.Methn* with *PrmLT*. Note here that we do not specify the index of the parameter because the parameter is known by a name that is distinct from all those other parameters declared in the method.

To apply the FGT on the underlying system the following should hold.

- The parameter *Prmn* should be declared in the *PrmLT* of the method *Pn.Cn.Methn*.
- If *ParmALT* denotes the type list of the method *Methn* after deleting *Prmn,* then the method *Methn* with *ParmALT* may not be declared in the class *Pn.Cn* or in any of its ancestor classes.

FGTPrecondConj(deleteObject(*Pn, Cn, Attn, _, _, attribute*) **):-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*),	1. *Parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.*
not(existsObject(*Pn, Cn, Methn, ParmALT, method*)),	2. *Method Pn.Cn.Methn with ParmALT not declared in the system, where ParmALT is the type list of the method Methn after deleting Prmn.*
not(isInherited(*Pn, Cn, Methn, ParmALT, method*)).	3. *Method Pn.Cn.Methn with ParmALT not declared in any of Pn.Cn's ancestor classes, where ParmALT again is the type list of the method Methn after deleting Prmn.*

4.2.2 Relational Element FGTs

This group of FGTs includes all FGTs that are used to modify *relational* elements in the system. (Recall that *relational* elements in the simplified UML meta-model include generalizations (extends), associations, reads, writes, calls and types.) The *relational* elements represent the relations that exist between two *object* elements. By using these FGTs, the developer can add, rename and delete *relational* elements that may exist between the *object* elements.

There are two types of *relational* elements. The first type includes those relations that are appeared in the class diagram and represent the relations between the different classes in the class diagram, like *extends* and *association relational* elements. The second type includes those relations that are found between the different *object* elements but are not represented in the UML class diagram. (For a detailed explanation return to chapter 3.)

4.2.2.1 addRelation FGT

The addRelation FGT is used to add a *relational* element between two different *object* elements in the UML class diagram. It is used to add *extends*, *association*, *read*, *write*, *call*, or *type relation* between two different *object* elements in the class diagram.

In general, it takes the following format:

addRelation(*RelL, SourceObject, DestinationObject, RelT*)

Where:

- *RelL* is the label of the *relation*. It is used when the *relation* to be added is an *extends* or *association* relation. It is ignored for the other types of relations.
- The *SourceObject* specified by the following parameters:
    - *FPn* is the name of the package of the source *object*. It is used when the source *object* is a package, class, method, attribute or parameter.
    - *FCn* is the name of the class of the source *object*. It is used when the source *object* is a class, method, attribute or parameter.
    - *FMemn* is the name of the member (method or attribute) of the source *object*. It is used when the source *object* is a method, attribute or parameter.
    - *FPrmn* is the name of the parameter. It is used when the source *object* is a parameter.
    - *FPrmLT* is the type list of a method's parameters. It is used when the source *object* is a method or a parameter.

  - *FOT* is the type of the source *object* (*class, method, attribute,* or *parameter*).
- The *DestinationObject* specified by the following parameters: *TPn, TCn, TMemn, TPrmn, TPrmLT,* and *TOT*. Each of these parameters has the same description as its associated parameter in the *SourceObject* as defined above.
- *RelT* is the type of the *relation* intended to be added between the source and destination *objects* (*extends, association, read, write, call or type*).

The set of arguments and precondition conjuncts that are used for the FGT addRelation are dependent on the type of *relation* that is to be added between the two *object* elements in class diagram, as shown below:

A. addRelation(*_, FPn, FCn, FMethn, _, FPrmLT, method, TPn, TCn, TAttn, _, _, attribute, RelT* **)** where *RelT* is *read* or *write.*

This FGT is used to add a *read/write* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the attribute *TPn.TCn.TAttn*. The *relation* between the method *FMethn* and the attribute *TAttn* indicates that at the code-level there will be one or more statements in the method *FMethn* that will *read/write* the attribute *TAttn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

To apply this FGT on the underlying system the following should hold.

- The source and the destination *objects* of the *relation* should be declared in the system.
- The relevant *relation* between the two *objects* is not already present.
- The location of the source *object* and the destination *object* in the model together with the access mode of the destination *object* play an important role in determining the applicability of the addRelation FGT. For example, if the access mode of the attribute *TAttn* is *private* then the method and the attribute should be allocated in the same class. This option and the other options are clarified by the conditions from 4 to 7 in the following list of precondition conjuncts.

FGTPrecondConj(addRelation(*_, FPn, FCn, FMethn, _, FPrmLT, method, TPn, TCn, TAttn,_,_, attribute, RelT*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.*
existsObject(*TPn, TCn, TAttn, attribute*),	2. *Attribute TPn.TCn.TAttn declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TAttn, attribute, RelT*)) ,	3. *Relation RelT not found between the two objects, the relation label is execluded from the condition.*
[(objectAMode(*TPn, TCn, TAttn, attribute, private*), *FPn.FCn=TPn.TCn*) |	4. *If the access mode of the attribute TAttn is private then the method FMethn should be in the same class of the attribute.*
(objectAMode(*TPn, TCn, TAttn, attribute, default*), *FPn=TPn*) |	5. *If the access mode of the attribute TAttn is default then the method FMethn should be in the same package of the attribute.*
(objectAMode(*TPn, TCn, TAttn, attribute, protected*), (subClass(*FPn,FCn, TPn, TCn*) | *FPn=TPn*)) |	6. *If the access mode of the attribute TAttn is protected then the method FMethn should be in the same package of the attribute or in one of the subclasses of the class TPn.TCn.*
objectAMode(*TPn, TCn, TAttn, attribute, public*)].	7. *If the access mode of the attribute TAttn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

A. addRelation(*_, FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call* **)**

This FGT is used to add a *call* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the method *TPn.TCn.TMethn* with *TPrmLT*. The *relation* between the two methods *FMethn* and *TMethn* indicates that at the code-level, there will be one or more statements in the method *FMethn* that will *call* the method *TMethn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(*_, FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.*
existsObject(*TPn, TCn, TMethn, TPrmLT, method*),	2. *Method TPn.TCn.TMethn with TPrmLT declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call*)),	3. *Relation call not found between the two objects.*
	4. *Note: we exclude the relation label from the condition.*
[(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, private*), *FPn.FCn=TPn.TCn*) |	5. *If the access mode of the method TMethn is private then the calling method FMethn should be in the same class of TMethn.*
(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, default*), *FPn=TPn*) |	6. *If the access mode of the method TMethn is default then the method FMethn should be in the same package of TMethn.*
(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, protected*), (subClass(*FPn,FCn, TPn, TCn*) | *FPn=TPn*)) |	7. *If the access mode of the method TMethn is protected then the method FMethn should be in the same package of the TMethn or in one of the subclasses of the class TPn.TCn.*
objectAMode(*TPn, TCn, TMethn, TPrmLT, method, public*)].	8. *If the access mode of the method TMethn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

C. addRelation(_, *FPn, FCn, FMethn,_, FPrmLT, method, TPn, TCn, _, _, _, class, type* **)**

This FGT is used to add a *type* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the class *TPn.TCn*. The *relation* between the method *FMethn* and the class indicates that at the code-level, the method *FMethn* defines at least one local variable whose type definition is class *TCn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn, _, FPrmLT, method, TPn, TCn, _, _, _, class, type*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Class TPn.TCn declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn,class, type*)),	3. *Relation type not found between the two objects.*
[(objectAMode(*TPn, TCn, class, default*), *FPn=TPn*) |	4. *If the access mode of the class TCn is default then the method FMethn should be in the same package of TCn.*
objectAMode(*TPn, TCn, class, public*)].	5. *If the access mode of the class TCn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

D. addRelation(_, *FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, type*)

This FGT is used to add a *type* relation between the class *FPn.FCn* and the class *TPn.TCn*. The *relation* between the two classes indicates that at the code-level, the class *FCn* defines an attribute of type class *TCn*. Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(_, *FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, type*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
[(objectAMode(*TPn, TCn, class, default*), *FPn=TPn*) |	3. *If the access mode of the class TCn is default then the class FCn should be in the same package of TCn.*
objectAMode(*TPn, TCn, class, public*)].	4. *If the access mode of the class TPn.TCn is public then the class FCn can be anywhere "the condition between the two brackets is true".*

E. addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, extends*)

This FGT is used to add an *extends relation* (*generalization/specialization*) with label *RelL* between the two classes *FPn.FCn* and *TPn.TCn*. The source *object* of the *relation FPn.FCn* will be the superclass while the destination *object TPn.TCn* will be the subclass.

To apply this FGT on the underlying system the following should hold.

- The source and the destination *objects* of the *relation* should be declared in the system.
- The *extends* relation between the two *objects* may not already exist.
- To avoid multiple inheritances between classes, the class *TCn* may not already be a subclass of any other class.
- To avoid circular *extends-relations* between classes, the class *FCn* should not be one of the descendants of the class *TCn.*

FGTPrecondConj(addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, extends*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
not(existRelation(_,*FPn, FCn, class, TPn, TCn, class, extends*)),	3. *Relation extends not found between the two classes*
not(subclass(*TPn, TCn*,_,_)),	4. *Class TPn.TCn is not a subclass of any other classes. This condition is to avoid multiple inheritance.*
not(subclass(*FPn, FCn*, *TPn, TCn*)).	5. *Class FPn.FCn is not one of the descendants of the TPn.TCn. This condition is to avoid circular extends between the classes.*

F. addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, association*)

This FGT is used to add an *association relation* with label *RelL* between the two classes *FPn.FCn* and *TPn.TCn*. The first class *FPn.FCn* will be the source of the *relation* while the second class *TPn.TCn* will be the destination of the *relation*. Note that if there is any *read*, *write*, or *type relation* between the class *FCn* (or any of its members) and the class *TCn* (or any of its members) then there should be an *association* relation from class *FCn* to class *TCn*.

FGTPrecondConj(addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, association*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
not(existRelation(*RelL, FPn, FCn, class, TPn, TCn, class, association*)).	3. *Relation association with label RelL not found between the two classes.*

4.2.2.2 renameRelation FGT

This FGT is used to change the label of the *relation* that exists between two *objects* from an old label *RelOldL* to a new one *RelNewL*. It has the following format:

renameRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT, RelNewL*)

Since the label of *extends* and *association* relations appear in the class diagram, this FGT can be used to change the label of the *extends* and *association* relations.

To apply this FGT on the underlying system the following should hold.

- The *relation* with label *RelOldL* and type *RelT* should already be found between the two object elements.
- The *relation* with label *RelNewL* and type *RelT* should not be found between the two object elements.

FGTPrecondConj(renameRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT, RelNewL*) **):-**

existRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT*)	1. *A relation with the old label RelOldL already exists between the two objects.*
not(existRelation(*RelNewL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT*))	2. *A relation with the new label RelNewL should not be found between the two objects.*

4.2.2.3 deleteRelation FGT

The deleteRelation FGT is used to delete a *relation* element that may exist between two different *object* elements in the class diagram.

A. deleteRelation(*_, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT, TOT, RelT* **)** where *RelT* is *read, write, call,* or *type.*

This FGT is used to delete a *relation* between two *objects*. However, the *relation* has to be *read, write, call* or *type.*

FGTPrecondConj(deleteRelation(_, *FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT*) **):-**

existRelation(_, *FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT*).	1. *The relation RelT between the two objects already exists in the system.*

B. deleteRelation(*RelL, FPn, FCn, _, _, _ ,class, TPn, TCn, _, _, _, class, association*)

This FGT is used to delete an *association* relation that may exist between two classes in the class diagram.

FGTPrecondConj(deleteRelation(*RelL, FPn, FCn, _, _, _,class, TPn, TCn, _, _, _,class, association*) **):-**

existRelation(*RelL, FPn, FCn, _, _, _ ,class, TPn, TCn, _, _, _,class, association*)	1. *The relation between the two classes has already existed in the system.*

C. deleteRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, extends*)

This FGT is used to delete an *extends* relation (*generalization/specialization*) with label *RelL* that is found between the two classes *FPn.FCn* as the superclass and *TPn.TCn* as the subclass.

To apply this FGT on the underlying system, the *relation* between the two classes has to be already found between the two classes. Furthermore, instances of class *TCn* (or any of *TCn's* descendant) may not reference any member which is inherited from the class *FCn* (or any of its ancestors). Clearly, if such a reference to such a member is found, then after deleting the *extends* relation that member will not be accessible to instances of *TCn* and its descendants. Trying to reference the member will therefore cause an error. This is checked by condition two of the following list of precondition conjuncts.

FGTPrecondConj(deleteRelation(*RelL, FPn, FCn,_, _, _, class, TPn, TCn, _, _, _, class, extends*) **):-**

existRelation(*RelL, FPn, FCn, FPrmLT ,class, TPn, TCn, TPrmLT ,class, extends*)) ,	1. *The relation between the two objects has already existed in the system.*

[existsObject(FPn, FCn, FMemn, FPrmLT, member), (objectAMode(FPn, FCn, FMemn, FPrmLT, member, OAMode), isElement(OAMode, [protected, public]), not(useInheritenceMem(TPn, TCn, FMemn,FPrmLT , member))], [existsObject(FPn, FCn, FMemn, FPrmLT, member), objectAMode(FPn, FCn, FMemn, FPrmLT, member, OAMode), isElement(OAMode, [protected, public]) , subclass(TTPn, TTCn, TPn, TCn) ,not(useInheritenceMem (TTPn, TTCn, FMemn , FPrmLT, member))], [supclass(FFPn, FFCn, FPn, FCn), existsObject(FFPn, FFCn, FMemn, FPrmLT, member) ,objectAMode(FPn, FCn, FMemn, FPrmLT, member, OAMode), isElement(OAMode, [protected, public]), not(useInheritenceMember(TPn, TCn, FMemn, FPrmLT, member))], [supclass(FFPn, FFCn, FPn, FCn), existsObject(FFPn, FFCn, FMemn, FPrmLT, member), objectAMode(FPn, FCn, FMemn, FPrmLT, member, OAMode), isElement(OAMode, [protected, public]) , subclass(TTPn, TTCn, TPn, TCn), not(useInheritenceMember (TTPn, TTCn, FMemn, FPrmLT, member))].	2. *If the class FCn or any of its superclasses "ancestors" have a member X and at the same time the member X is inherited and used from outside through instances of the class TCn or any of its descendants then the extends relation between the two classes (FCn and TCn) cannot be deleted.*

4.3 FGT Sequential Dependency

In the foregoing, the notion of a postcondition of an FGT has not been discussed. Nevertheless, it is evident that whenever an FGT is applied to a system, one or more of its precondition conjuncts will be negated. For example:

- In adding an object, the precondition that the object may not exist is negated, and the object now exists in the system.

- In renaming an object, the precondition requires that an object of the old name may exist, and an object of the new name may not exist.

After application of the relevant FGT, then requirements are negated. The conjunction of these negated precondition conjuncts after applying an FGT will be considered to be its postcondition conjuncts.

4.3.1 Definition

FGT_j is said to be potentially sequentially dependent on FGT_i if and only if the postcondition conjuncts of FGT_i satisfied one or more precondition conjuncts of FGT_j. The sequential dependency between the two FGTs is represented by:

$$FGT_i \rightarrow FGT_j$$

For example, the FGT

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

that is used to add the method *m1* in the class *P.A* is sequentially dependent on the FGT

addObject(*P, A, _, _, _, _, public, _, class*)

that is used to add the class *A* in the package *P*, because one first has to add the container (class *P.A*) before adding members in it. The sequential dependency between the two FGTs is represented as:

addObject(*P, A, _, _, _, _, public, _, class*) → addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

Note that, as defined above, the potential sequential dependency between two FGTs does not depend on the description of the system under consideration. (This is in contrast with Robert [70], who defines sequential dependency between two refactorings relative to program or system.) This means that if $FGT_j \rightarrow FGT_i$ and there is a need to apply FGT_i to some given system, S, one of the following scenarios may occur:

- S is such that it already satisfies all precondition conjuncts of FGT_i. Thus, FGT_i may be directly applied to S. In this case, it would not be possible to apply FGT_j to S, since the satisfaction of all FGT_i's precondition conjuncts indicates that at least one of FGT_j's precondition conjuncts is not satisfied by S.

- S does not satisfy all precondition conjuncts of FGT$_i$. In this case it will be necessary to select one or more FGTs upon which FGT$_i$ sequentially depends, to apply it to S, and then to apply FGT$_i$. Whether or not FGT$_j$ is to be included in this selection depends on S.

All the potential sequential dependencies between all the FGTs in the earlier sections of this chapter have been catalogued. These are shown in Figure 4.1. The figure shows two types of sequential dependency between the different FGTs. Each one of the two types will be explained in detail in the following two subsections.

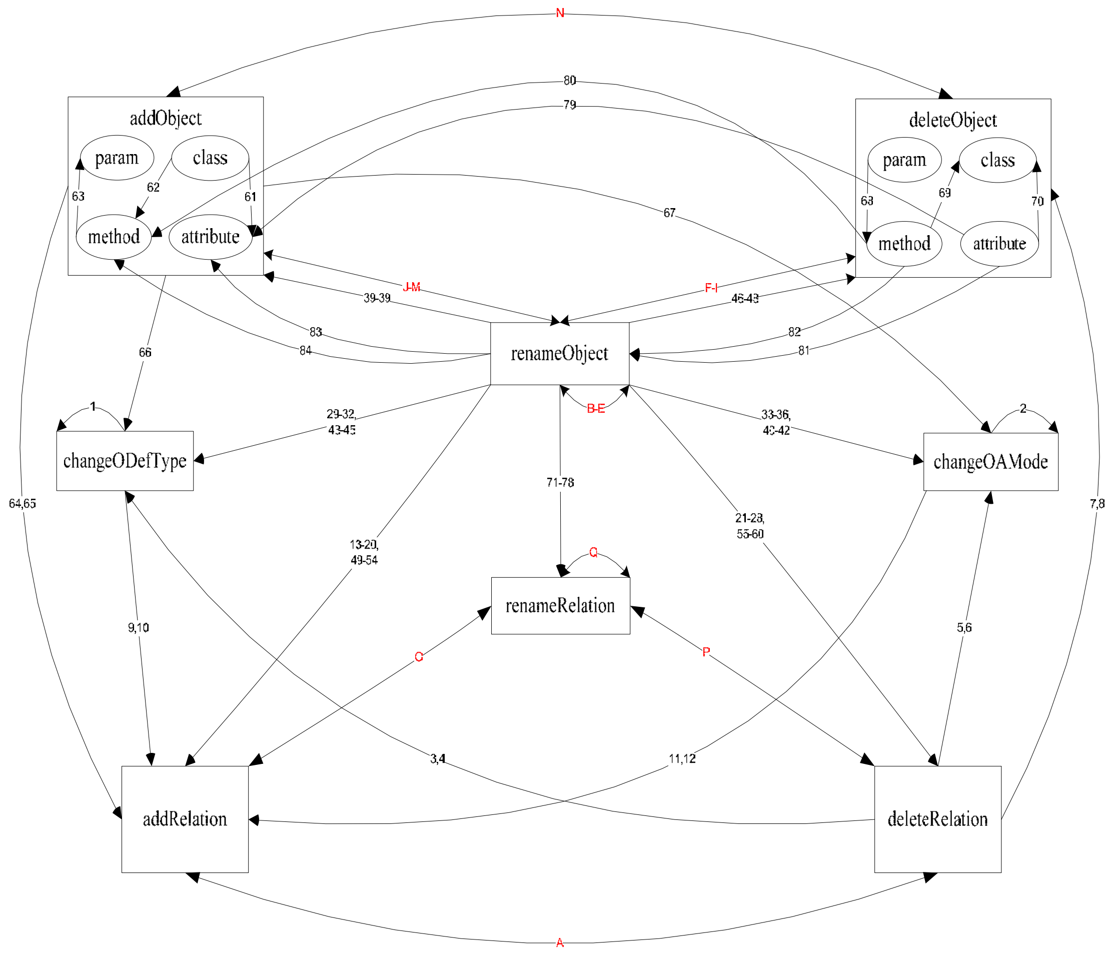


Figure 4.1: Potential sequential dependencies between FGTs

4.3.2 Uni-Directional Sequential Dependencies

The first kind includes sequential dependencies that occur in one direction between the two FGTs (*FGT$_i$* and *FGT$_j$*). This means that *FGT$_j$* is sequentially dependent on *FGT$_i$* but *FGT$_i$* is not sequentially dependent on FGT_j. The uni-directional sequential dependency between FGTs is represented as arcs with a single arrow at one end in Figure 4.1. This category of sequential dependencies is represented as **uniDirSD** facts in the Prolog database of the refactoring tool.

All the uni-directional sequential dependencies in Figure 4.1 are discussed in more detail in Appendix A.1. It will be seen that each numbered sequential dependency corresponds to a numbered arc in Figure 4.1 that represents a uni-directional sequential dependency.

For example, consider the following FGTs:

FGT_i : addObject(*P, C, _, _, _, _, public, _, class*) and

FGT_j : addObject(*P, C, att1, _, _, type(basic, int, 0), private, _, attribute*)

The information represented in Figure 4.1 (the arrow labelled 61) shows that FGT_j is sequentially dependent on FGT_i ($FGT_i \rightarrow FGT_j$). This is because the class *P.C* has to be added to the system first. Only after that can the attribute *att1* be added in that class. On the other hand, FGT_i is not sequentially dependent on FGT_j.

4.3.3 Bi-Directional Sequential Dependency

The second type of sequential dependency includes sequential dependencies that can occur in the two directions of the two FGTs. This means that the first FGT is sequentially dependent on the second one and that the second FGT is sequentially dependent on the first one—i.e. for FGT_i and FGT_j, $FGT_i \leftrightarrow FGT_j$. The bi-directional sequential dependencies are represented as arcs with arrows at both ends in Figure 4.1. This category of sequential dependencies is represented as **biDirSD** facts in the Prolog database of the refactoring tool.

All the bi-directional sequential dependencies in Figure 4.1 are discussed in more detail in Appendix A.2. It will be seen that each numbered sequential dependency corresponds to a numbered arc in Figure 4.1 that represents a bi-directional sequential dependency.

For example, consider the following FGTs:

FGT_i: addObject(*P ,A, f1, _, _, type(basic, int, 0), private, _, attribute*) and

FGT_j : renameObject(*P, A, f1, _, _, attribute, f2*)

Two forms of sequential dependencies can occur between the two FGTs in this example:

a. $FGT_i \rightarrow FGT_j$: This is the case when class *A* does not contain attribute *f1*. The attribute *f1* then has to be added in class *A* by the addObject FGT. Thereafter, the added attribute can be renamed from *f1* to *f2* by the renameObject FGT. Thus, here the renameObject FGT is sequentially dependent on the addObject FGT.

b. $FGT_j \rightarrow FGT_i$: This is the case when attribute *f1* is originally declared in class *A* so adding another attribute with same name *f1* will cause duplication. Here the renameObject FGT has to be used to change the name of *f1* to *f2* and thereafter the addObject FGT can be used to add the attribute *f1* in class *A*. In this case the FGT addObject is sequentially dependent on the renameObject FGT.

To decide which sequential dependency applies in a given situation, the state of the underlying system has to be taken into consideration. This will be discussed in more detail in chapter 9.

4.3.4 Mapping Feasible FGT-Lists to FGT-DAGs

This section takes as a starting point a **feasible FGT-list**. By this is meant a list of FGTs for which at least one system exists, such that the FGT elements in the list can feasibly be applied to the system, starting at the head of the list and applying each successive FGT until the tail of the list has been applied. A consequence of applying the list to an appropriate system is that a set of objects and a set of relations (each possibly empty) will be guaranteed to exist in the system; and a set of objects and a set of relations (each possibly empty) will be guaranteed *not* to exist in the system. The conjunction of the assertions about the existence and non-existence of these entities can be regarded as the list's postcondition.

Of course, not every list of FGTs is feasible. For example, any FGT-list that specifies two successive deletions of the same object cannot be feasible, since the precondition of the second—the object's existence—cannot be met. Nevertheless, for the purposes of describing the algorithm given later in this section, the origin of such a feasible list of FGTs is currently not of concern. It may, for example, be an FGT-list proposed by a developer who wishes to transform a given system design in some particular way. The transformation may or may not retain the original system behaviour—i.e. it may or may not be a refactoring.

This section is also concerned with the notion of an **FGT Directed Acyclic Graph (FGT-DAG)**. An FGT-DAG is a directed acyclic graph in which each node represents an FGT, and there is an arc between two nodes, say from node FGT_j to FGT_i, if and only if:

1. FGT_i is sequentially dependent on FGT_j;

2. FGT_i is not sequentially dependent on any successor of FGT_j; and

3. no ancestor of FGT_j has an arc to FGT_i (even if FGT_i is sequentially dependent on that ancestor).

An FGT-DAG is feasible if some system exists to which it can feasibly be applied. An FGT-DAG is applied to a system by applying the FGTs in any order that respects the sequential dependency relationships represented by the arcs. This means that an FGT may only be applied after all its ancestors have been applied. As in the case of a feasible FGT-list, a feasible FGT-DAG is characterised by a postcondition—the conjunction of predicates asserting what objects and relations exist and/or do not exist as a result of applying the FGT-DAG. Similarly, the postcondition of a set of feasible FGT-DAGs is simply the conjunction of the postconditions of its constituent FGT-DAGs, and is attained by applying these in any order.

Clearly, if a feasible FGT-list is to be applied to some system, the system should comply with certain requirements that ensure that the FGTs in the list can indeed be applied in the given order—i.e. the feasible FGT-list has a certain precondition conjuncts to which the system should conform. The conjuncts of the precondition of this feasible FGT-list are not simply the conjuncts of all precondition conjuncts of its constituent FGTs. Indeed, it consists of the conjunction of FGT precondition conjuncts that are not negated as a result of applying the FGTs. For example, consider the feasible FGT-list [FGT_1, FGT_2]. Suppose the precondition of FGT_1 is P_1^P_2 and the precondition of FGT_2 is P_3^P_4. Suppose, also, that the postcondition of FGT_1 is P_3 (or, more generally, that it logically implies P_3, but not P_4). Then the precondition of the list is P_1^P_2^P_4. By similar argumentation, a set of feasible FGT-DAGs also has a precondition.

In the remainder of this thesis, it should be assumed that the reference to an FGT-list or set of FGT-DAGs would be taken to mean a **feasible** FGT-list or set of FGT-DAGs, unless otherwise stated. Furthermore, a sequence of FGTs should be regarded as equivalent to a list of FGTs, the latter simply indicating the concrete implementation of a sequence in the Prolog context.

The question then arises: How can a feasible FGT-list can be mapped to a set of FGT-DAGs that has the same postcondition as the feasible FGT-list? An algorithm, called **build-FGT-DAG** has been implemented in the prototype tool to do that. Algorithm 4.1 provides the pseudo-code for the build-FGT-DAG algorithm. The algorithm derives from a feasible FGT-list, *FGTList,* a set of FGT-DAGs, DSET, that has the same postcondition as *FGTList*.

It does this by setting DSET to the empty set, and then processing the FGTs in *FGTList* from first to last. Each next FGT, FGT_i, to be processed begins as a new singleton FGT-DAG in DSET. All paths in the other FGT-DAGs in DSET are then traversed in a bottom-up fashion, searching for the first FGT upon which FGT_i sequentially depends. If such a node, FGT_j, is found in a path, it is connected to FGT_i and all ancestors of FGT_j are eliminated as candidates for further consideration.

Algorithm 4.1 (Building FGT-DAGs algorithm)

```
build-FGT-DAG ( FGTList )
Input:    FGTList: A feasible list of FGTs
          uniDirSD: Uni-directional sequential dependencies between FGTs
          biDirSD: Bi-directional sequential dependencies between FGTs

Output:   DSET: A set of FGT-DAGs whose postcondition is the same as that of FGTList

Set DSET to the empty set
For each $FGT_i$ in FGTList do {  //FGTs should be selected in order from first to last
    Mark each FGT in each FGT-DAG of DSET as unchecked
    Insert $FGT_i$ into DSET as a single node of a new FGT-DAG and mark it as checked
    While there are unchecked FGTs in DSET do {
        Select an unchecked FGT with no unchecked children, say $FGT_j$
        Mark $FGT_j$ as checked
        If  $FGT_j \rightarrow FGT_i$ (as determined from uniDirSD and biDirSD) then {
            Mark all ancestors of $FGT_j$ as checked
            Insert an arc from $FGT_j$ to $FGT_i$
        } //enf If
    } //end While
}//end For
Return DSET
```

The algorithm will build the same set of FGT-DAGs from a given feasible FGT-List. Firstly, note the comment in the For-each loop: FGTs are selected in the order in which they appear in the list. Secondly, note that there is no possibility of non-determinism because of the potential alternative selections in the While-loop. To see this, suppose that FGT_j and FGT_k are both candidates for selection as FGTs with no unchecked children. If FGT_j is selected before FGT_k, then a link may (because of sequential dependency) or may not be established from FGT_j to FGT_i. However, it can easily be seen that this selection will not cause FGT_k to be checked. Instead, FGT_k will then be a candidate for selection in the next iteration.

When the algorithm completes, each FGT in *FGTList* will have been inserted into one and only one FGT-DAG in the set of FGT-DAGs. Each FGT node will have inbound arcs from the closest FGTs that precede it in *FGTList* upon which it sequentially depends. It will have outbound arcs to the closest FGTs following it in *FGTList* and which are sequentially dependent on it. Note that the algorithm has been designed to ensure that whenever a candidate bi-directional sequential dependency relationship between two elements in *FGTList* is found, the direction reflected in the FGT-DAG corresponds to the intended execution order dictated by *FGTList*.

Note that the structure (FGT-DAGs) produced by the algorithm are indeed acyclic, and not cyclic. This can be verified by considering the following two points:

a. The resulting set of DAGs represents a feasible FGT list. As suggested at the beginning of this section, the FGTs of a feasible FGT list are ordered according to their sequential dependencies in such a way that their overall precondition does not evaluate to false.

b. Logically, the set of DAGs have been set up in such a way that they encapsulate the sequential dependency between the FGTs. The sequential dependency conveys the nature of the pre/post conditions of the FGTs. Suppose that one FGT-DAG contained a cycle of sequential dependencies, for example: A→B→C and C→A. This would mean that part of the post conditions of C is needed to satisfy the preconditions of A and at the same time part of the postcondition of A is needed to satisfy the preconditions of C, which leads to a contradiction. Such a contradiction could only arise if the input FGT list was not feasible.

As a toy example, Figure 4.2 shows the FGT-DAGs that are produced for the following collection of FGTs of a refactoring X. The result shows that the FGTs of refactoring X are allocated inside three different FGT-DAGs which are sequentially independent:

- renameObject(*lan, A, _, _, _, class, B*),
- addObject(*lan, C, t, _, _, _, public, [], method*),
- addObject(*lan, B, y, _, _, _, public, _, attribute*),
- renameObject(*lan, B, y, _, _, attribute, x*),
- changeODefType(*lan, B, x, _, _, attribute, int, float*),
- deleteObject(*lan, S, m, _, [], method*),
- addObject(*lan, S, m, _, _, _, private, [], method*),
- changeOAMode(*lan, S, m, _, [], method, private, public*),
- deleteObject(*lan, Super, x, _, _, attribute*),
- renameObject(*lan, C, m, _, [], method, n*),
- addRelation(*l, lan, C, n, _, [], method, lan, S, m, _, [], method, call*),

- deleteRelation(*l, lan, C, n, _, [], method, lan, S, m, _, [], method, call*),
- deleteObject(*lan, C, n, _, [], method*),
- renameObject(*lan, C, t, _, [], method, h*),
- addRelation(*l1, lan, S, m, _, [], method, lan, B, x, _, _, attribute, write*).

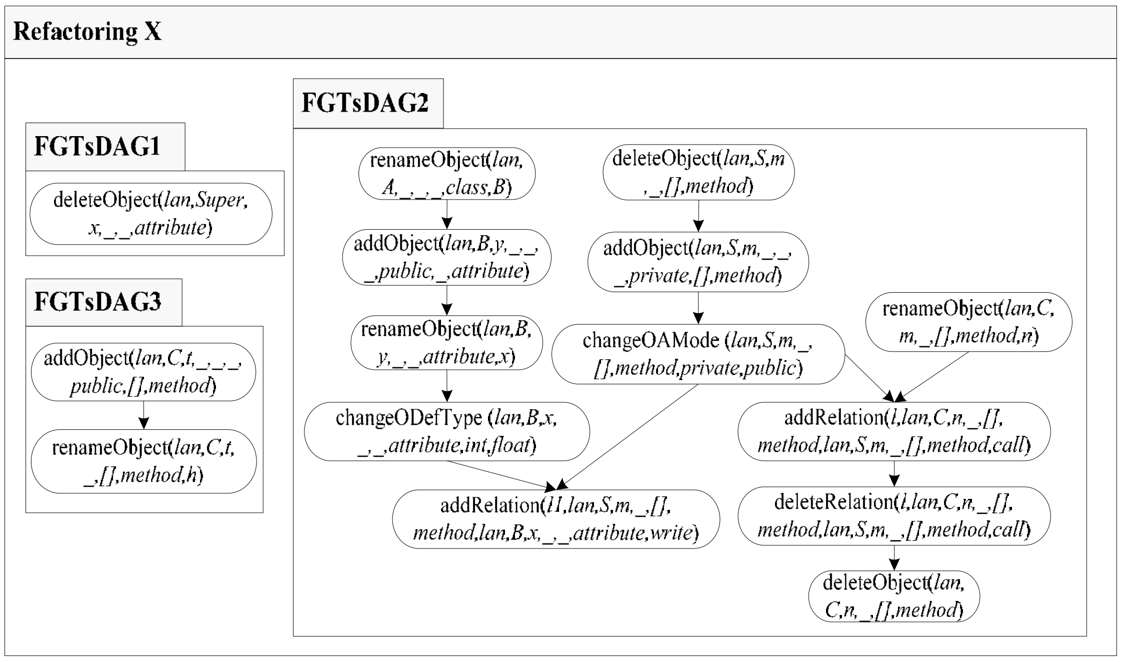

Refactoring X
FGTsDAG1
deleteObject(lan,Super, x,_,_,attribute)
FGTsDAG3
addObject(lan,C,t,_,_,_, public,[],method)
renameObject(lan,C,t, _,[],method,h)
FGTsDAG2
renameObject(lan, A,_,_,_,class,B)
addObject(lan,B,y,_,_, _,public,_,attribute)
renameObject(lan,B, y,_,_,attribute,x)
changeODefType (lan,B,x, _,_,attribute,int,float)
deleteObject(lan,S,m ,_,[],method)
addObject(lan,S,m,_,_, _,private,[],method)
changeOAMode (lan,S,m,_, [],method,private,public)
renameObject(lan,C, m,_,[],method,n)
addRelation(l1,lan,S,m,_,[], method,lan,B,x,_,_,attribute,write)
addRelation(l,lan,C,n,_,[], method,lan,S,m,_,[],method,call)
deleteRelation(l,lan,C,n,_,[], method,lan,S,m,_,[],method,call)
deleteObject(lan, C,n,_,[],method)

Figure 4.2: FGT-DAGs of refactoring X

4.4 FGTs for Primitive and Composite Refactorings

This section discusses in overview how to deal with primitive and composite refactorings in terms of their transformation operations and their preconditions. In particular, the section shows the relationship between the previously identified set of FGTs and refactorings, whether primitives or composites. A more complete discussion of primitive refactorings is taken up in chapter 5, and of composite refactorings in chapter 10.

4.4.1 Definitions

Definition 1: A **primitive refactoring** is an atomic refactoring that cannot be split into more refactorings. In the refactoring literature, researchers agree that there exists a finite set of primitive refactorings [65, 70]. The list of primitive refactorings that are commonly agreed upon is shown in Table 4.1.

A primitive refactoring may be said to be sound if its application to a system, say S1, which complies with its precondition results in a system, say S2, whose behaviour is the same as that of S1. Of course, S1 and S2 have the same behaviour if and only if for all possible input their resulting output is the same.

Furthermore, the collection of primitive refactorings in Table 4.1 can be regarded as complete with respect to the FGTs in this thesis if and only if it is not possible to use some set these FGTs to define a new primitive refactoring, that has not been mentioned in Table 4.1.

The question may be asked whether the primitive refactorings in Table 4.1 are sound and complete. It is beyond the scope of this thesis to provide a formal answer to this question. For their soundness, we appeal to their appearance in the literature. Should they be incomplete (in the sense mentioned above), then, per definition, it will be possible to use FGTs to add to the menu of primitive refactoring given in the table.

For each primitive refactoring, a precondition exists that will guarantee behaviour preservation of the system. This precondition is implemented inside the refactoring tool and need to be checked before applying the related refactoring.

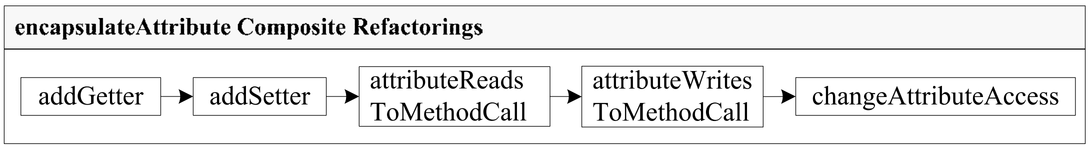
Table 4.1: Primitive refactorings

Add Element Refactorings	**Delete Element Refactorings**	**Change Element Refactorings**	
		Change Characteristics	**Change Structure**
addClass addMethod addAttribute addParameter	deleteClass deleteMethod deleteAttribute deleteParameter	renameClass renameMethod renameAttribute renameParameter	changeSuper
			moveMethod moveAttribute
addGetter addSetter		changeClassAccess changeMethodAccess changeAttributeAccess	attributeReadsToMethodCall attributeWritesToMethodCall
		changeMethodType changeAttributeType changeParameterType	pullUpMethod pullUpAttribute pushDownMethod pushDownAttribute

Definition 2: A **composite refactoring** is a collection of primitive refactorings that are applied on the model as one unit. In part of the composite refactoring, the execution order of some of, but not necessarily all, its primitive refactorings may be specified. Each composite refactoring has its own precondition. This precondition may not simply be the conjunction of its constituent primitive refactorings preconditions. Instead, it should articulate system conditions

that make it possible to apply the primitive refactorings in the order required by the composite refactoring.

An example of a composite refactoring is the **encapsulateAttribute** composite refactoring. This composite, as will be shown in more detail in chapter 6, consists of the following sequence of primitive refactorings:


encapsulateAttribute Composite Refactorings
addGetter
addSetter
attributeReads ToMethodCall
attributeWrites ToMethodCall
changeAttributeAccess

The current approach shifts the granularity of transformation one level down: primitive refactorings are constructed from a collection of FGTs ordered in FGT-DAGs. Thus, FGTs are the most fine-grained type of transformations under consideration. The relationship between primitive refactorings, composite refactorings and FGTs is intuitively reflected in Figure 4.3(b). Figure 4.3(a) shows that a composite refactoring is a collection of primitive ones, and each primitive refactoring can be defined as a collection of FGTs. Thus, each composite refactoring can be carried out as a collection of FGTs.

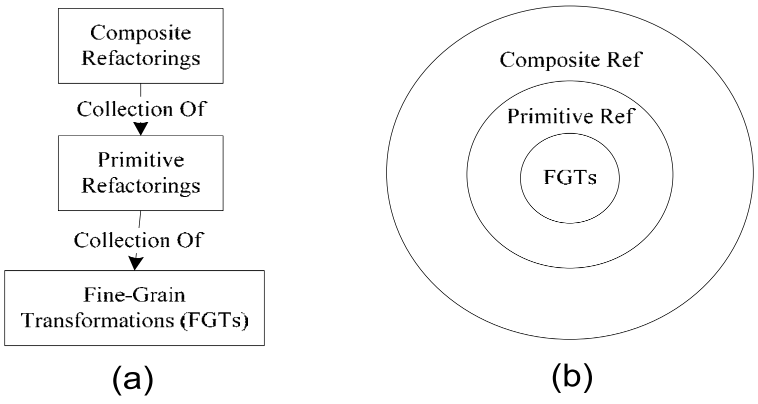


Figure 4.3: Primitive, composite refactorings and FGTs

4.4.2 FGT-Enabling Preconditions in an FGT-DAG

It is evident an FGT in an FGT-DAG has the property that the postcondition of each of its parents logically entails of one or more of that FGT's precondition conjuncts. If

$$Pre = \{P_i \mid i = 1, \ldots n\}$$

is the set of the FGT's precondition conjuncts, and

$$\text{Post} = \{Q_j \mid j = 1, \ldots m\}$$

is the set of postconditions of all of its parents, then

$$\text{En} = \{P_i \in \text{Pre} \mid \forall\, Q_j \in \text{Post} : \sim(Q_j \Rightarrow P_i)\}$$

defines the set of precondition conjuncts of the FGT that are not entailed by its parents' postconditions. This will be called the FGT's set of **enabling precondition conjuncts**.

The union of the enabling precondition conjuncts of all FGTs in an FGT-DAG is the FGT-DAG's set of enabling precondition conjuncts. The conjunction of all enabling precondition conjuncts of all the FGT-DAGs in a refactoring is called the **FGT-enabling** precondition of the refactoring (and also of the set of FGT-DAGs).

Clearly, if a system complies with the FGT-enabling precondition of an FGT-DAG, then the FGTs in the FGT-DAG can be systematically applied to the system in the order determined by the FGT-DAG, with the assurance that all FGT preconditions will be fulfilled by the system when they are to be applied to the system.

4.4.3 FGTs and Primitive Refactorings Preconditions

In much of the literature on refactoring, precondition conjuncts for each respective primitive refactoring are specified. Figure 4.4(b) shows how the current refactoring approaches deal with such refactorings preconditions. All the precondition conjuncts of the refactoring in these approaches are installed as one unit at the level of the whole refactoring. If the system complies with all of the primitive's precondition conjuncts, then the refactoring is applied to the system, and behaviour is guaranteed to be preserved. (Note that this application occurs "atomically", which is why Figure 4.4(b) represents the refactoring as a black box.)

However, a primitive refactoring can be represented as a collection of FGTs ordered within a set of FGT-DAGs. This will be discussed in chapter 5. As seen in section 4.4.2, associated with each FGT-DAG is a specific FGT-enabling precondition. This raises the following question:

Suppose that a primitive refactoring is represented as a set of FGT-DAGs. Will behaviour of a system be preserved if all the FGT-enabling preconditions of all the FGT-DAGs are satisfied before their individual FGTs are applied (in the appropriate order) to the system? The answer is NO. To justify this claim consider the following point.

For some primitive refactorings, there are special precondition conjuncts that cannot be inferred from the precondition conjuncts of the FGTs included in the primitive refactoring. For example, one of the precondition conjuncts of the refactoring **pullUpAttribute** that is used to pull up an attribute *Attn* to the superclass from all subclasses where it is defined, is that the attribute *Attn* should be declared identically (have the same definition type) in all the subclasses where it is defined. Consideration of the FGTs used in such a **pullUpAttribute** refactoring (not given here, but in chapter 5) will show that it is impossible to infer such a precondition conjunct from the preconditions of the included FGTs.

In general, it is therefore necessary to isolate the set of precondition conjuncts of a primitive refactoring that are not logically entailed by any of the FGT-enabling preconditions of the FGT-DAGs from which the primitive refactoring is constructed. These isolated conjuncts will be referred to as **refactoring-level precondition conjuncts**. In principle, therefore, a refactoring that is specified as a set of FGT-DAGs will preserve a system's behaviour if the system initially complies with the refactoring-level precondition conjuncts, and also complies with the FGT-enabling preconditions of all FGT-DAGs in the set. Figure 4.4(a), which shows the FGT-DAGs for a fictitious primitive refactoring, thus also portrays the refactoring-level precondition, as well as FGT-enabling preconditions.

In the present text, the focus is on precondition conjuncts. However, postconditions can also be viewed as being at the refactoring-level as well as at the FGT-level. These notions are abstractly portrayed in Figures 4.4(a) and 4.4(b) with respect to an FGT approach and previous approaches respectively.

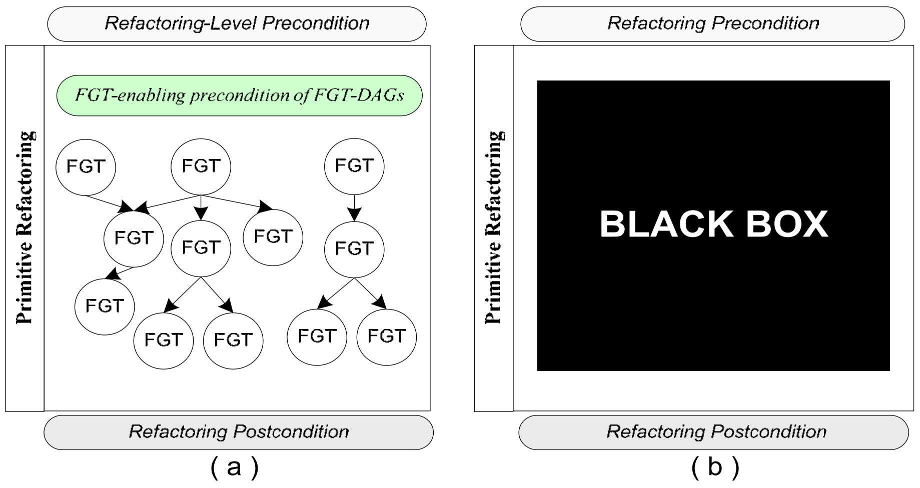

Refactoring-Level Precondition
FGT-enabling precondition of FGT-DAGs
Primitive Refactoring
FGT
Refactoring Postcondition
(a)
Refactoring Precondition
Primitive Refactoring
BLACK BOX
Refactoring Postcondition
(b)

Figure 4.4: Primitive refactoring different considerations

4.4.4 Applying Refactorings

Using a tool to apply a specific refactoring to the system is done in two phases: in the first phase, the tool has to check both the refactoring-level precondition as well as all the FGT-enabling preconditions of the various FGT-DAGs. If these are satisfied then it proceeds to the second phase in which the refactoring itself is applied to the system—i.e. the tool's code that updates the tool's representation of the UML model.

Dealing with two levels of precondition conjuncts introduces the followings themes:

a. As explained in chapter 7, when an FGT is cancelled or absorbed by the reduction process, then its set of precondition conjuncts will also be cancelled or absorbed, which means that the overall number of refactoring precondition conjuncts may potentially be reduced. The overall effect will be to reduce the number of precondition conjuncts that need to be checked, potentially enhancing the performance of the refactoring tool.

b. Consideration should be given to the parallelizing opportunity at the time of checking the precondition conjuncts of FGTs and also at the time of applying that FGTs. (Addressed in chapter 11)

c. Because the precondition conjuncts of the FGTs are predefining and pre-implemented in the refactoring tool, an end user of the refactoring tool who chooses to define a new refactoring merely has to be concerned with the precondition conjuncts at the refactoring-level. (Addressed in chapter 12)

4.5 Reflection on this Chapter

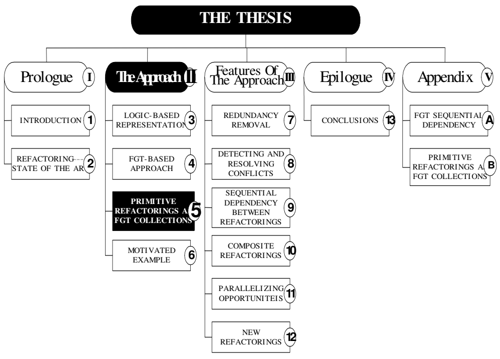
This chapter has introduced the notion of FGTs and catalogued those relevant to this thesis, together with their associated precondition conjuncts. It has suggested that a collection of such FGTs, ordered in a set of FGT-DAGs, can be used to transform a system. It has also suggested that where such transformations constitute a refactoring, certain refactoring-level precondition conjuncts can be isolated from FGT-level precondition conjuncts and be processed separately. The remainder of the thesis elaborates on the consequence of using FGTs in this fashion.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

CHAPTER 5

PRIMITIVE REFACTORINGS AS FGT COLLECTIONS

5.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
LOGIC-BASED REPRESENTATIO 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS A FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS A FGT COLLECTIONS B

This chapter elaborates on the feasibility of representing primitive refactorings as a collection of FGTs. The term "collection" is used here to designate either an FGT-list or an equivalent set of FGT-DAGs, as discussed in the previous chapter. Twenty-nine well-known primitive refactorings that are frequently defined and used in refactoring literatures will be introduced [22, 60, 65, and 66]. Each primitive refactoring is represented as a collection of FGTs instead of implementing it as a piece of code (*black box*). The chapter shows that some of these primitive refactorings can be represented by a single FGT while others need the application of several FGTs in an FGT-list. The chapter also discusses the relationship between the precondition conjuncts of the primitive refactorings and the precondition conjuncts of the associated FGTs.

The concern in this thesis is to propose a new approach to formalize model refactorings. It is beyond of the scope of this thesis to discuss in detail the theme of so-called ``*code-smells*``— i.e. to identify opportunities for refactoring in the system and to propose suitable refactorings to be use in the presence of such code-smells. This is, in fact, an entirely different area of research in the field of refactorings. There exist a number of detection tools that automatically detect opportunities for refactorings on the system [76, 85]. Such tools are based on various metrics of software quality and other techniques.

The primitive refactorings discussed in this chapter are categorized into three groups according to the kind of transformations they make on the underlying system: the first group, 'Add Element Refactorings', includes all refactorings that, when executed, will add elements to the system. These elements may be *object* or *relational* elements. The second group, 'Change Element Refactorings', includes all refactorings that, when executed, will change the characteristics of the element such as name, access mode or definition type. Alternatively, they will change the structure of the elements in the system by moving the element from one place to another. The third group, 'Delete Element Refactorings', includes all refactorings that will delete element or elements from the system under consideration. The list of primitive refactorings in each group is:

1. Add Element Refactorings
   a. addClass
   b. addMethod
   c. addAttribute
   d. addParameter
   e. addGetter
   f. addSetter
2. Change Element Refactorings

2.1 Change Characteristics
   a. renameClass
   b. renameMethod
   c. renameAttribute
   d. renameParameter
   e. changeClassAccess
   f. changeMethodAccess
   g. changeAttributeAccess
   h. changeMethodRetType
   i. changeAttributeType
   j. changeParameterType

2.2 Change Structure (Restructuring)
   a. changeSuper
   b. moveMethod
   c. moveAttribute
   d. attributeReadsToMethodCall
   e. attributeWritesToMethodCall
   f. pullUpMethod
   g. pushDownMethod
   h. pullUpAttribute ()
   i. pushDownAttribute

3. Delete Element Refactorings
   a. deleteClass

b. deleteMethod
c. deleteAttribute
d. deleteParameter

These twenty-nine primitive refactorings have been stored in the Prolog prototype tool as generic (i.e. uninstantiated) FGT-lists. In order to generate a particular refactoring that relates to elements in the system's representation of an UML class diagram, the name of the refactoring and its instantiated parameters are provided to the tool. The tool then instantiates the relevant stored FGT-list. Subsequently, the algorithm given in 4.3.4 may be used to build the corresponding set of FGT-DAGs. The refactoring may then be applied, and the system's representation will be changed accordingly.

In what follows, selected primitive refactorings and their mappings to FGTs will be discussed. In each case, the following headings will be used: Parameters; Description; Precondition Conjuncts; FGTs in the order of the stored FGT-list; followed by a note that relates the FGT and primitive refactoring precondition conjuncts. Primitive refactorings that map to a single FGT will not be given here, but—for completeness—will be discussed under the same headings in Appendix B.

In some cases, it will be seen that sequential compliance with the FGT precondition conjuncts that make up a primitive refactoring is sufficient to guarantee system behaviour as well. In this case, the FGT precondition conjuncts are said to cover the primitive refactoring precondition conjuncts.

Note that by sequential compliance is meant that the FGTs precondition holds at the point at which the FGT is about to be applied—not that the conjunction of all FGTs making up a primitive refactoring hold from the start. The claim that FGT precondition conjuncts cover the primitive refactoring precondition conjuncts is therefore not the same as the claim that the conjunction of FGT precondition conjuncts logically entails the precondition conjuncts of the associated primitive refactoring.

It will also be seen that in some cases, the precondition conjuncts of the FGTs do not cover the precondition conjuncts of the associated primitive refactoring. Mere compliance with FGT precondition conjuncts will therefore not necessarily guarantee behaviour preservation. In such cases, it is necessary to define so-called refactoring-level precondition conjuncts. To guarantee behaviour preservation, compliance with these should be checked before checking as mentioned in 4.4 and applying the constituent FGTs.

5.2 Add Element Refactorings

Refactorings in this group are used to add new elements to the software. The first four refactorings in the group are used to add class, method, attribute and parameter object elements to the system, while the last two refactorings are used to add getter and setter methods for specific attributes in the system. From a formal point of view, the first four refactorings are a behaviour-preserving—they do not change the behaviour of the system after refactoring—because none of the elements that they add are referenced in the system. The last two refactorings, addGetter and addSetter, are also behaviour-preserving because, even though the added methods (getter and setter) reference one of the existing attributes in the system, these methods (getter and setter) themselves are unreferenced from anywhere in the system.

5.2.1 addClass(*ClassName, AccessMode*)

The refactoring adds a new class to the system under consideration. The created class will be empty and standalone (no members, super or subclasses). (*For more details see Appendix B.1.1*)

5.2.2 addMethod(*MethodName, ReturnDType, AccessMode, ParameterList*)

The refactoring adds a new method in one of the classes of the system under consideration. (*For more details see Appendix B.1.2*)

5.2.3 addAttribute(*AttibuteName, AttributeDType, AccessMode*)

The refactoring adds a new attribute in one of the classes of the system under consideration. (*For more details see Appendix B.1.3*)

5.2.4 addParameter(*Prmname, PrmDType, Index, MethTList*)

The refactoring declares a new parameter in one of the methods of the system under consideration. (*For more details see Appendix B.1.4*)

5.2.5 addGetter(*AttributeName*)

Where *AttributeName* has the following format: *Pn.Cn.Attn* (*Pn* is the name of the package, *Cn* is the name of the class and *Attn* is the name of the attribute).

Description

The refactoring adds a getter method in the class *Pn.Cn*. This method is used to return (*get*) the value of the attribute *Attn* that is defined in the class *Pn.Cn*. Hence, the definition type of the return value of the getter method is the same as that of the attribute *Attn*.

Figure 5.1 shows the effect of the refactoring addGetter when it is applied to the *private* attribute *A.x* using: addGetter(*A.x*).

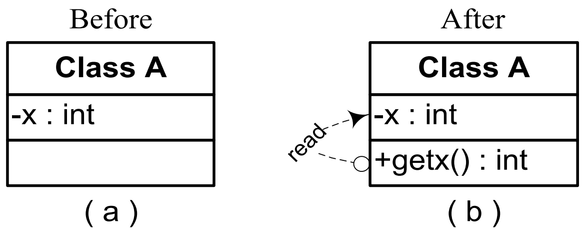


Figure 5.1: Class A before and after addGetter(*A.x*)

Precondition Conjuncts

(1) The signature of the getter method is distinct from those of all methods declared already in the class *Pn.Cn* and of any of its ancestors.

(2) The attribute *AttributeName* is declared in the class *Pn.Cn*.

FGT-List

1. addObject(*Pn,Cn, Methn,_,_,AttType ,public,[ ],method*)
2. addRelation(*_,Pn,Cn,Methn,_,[ ],method,Pn,Cn,Attn,_,_,attribute,read*)

Note

- FGT 1 in the FGT-list is used to add the getter method with no parameters. The name of the getter method *Methn* is formulated automatically by using the procedure **concat**(*'get', Attn, Methn*). The procedure concatenates the word *'get'* with the attribute *Attn*. The return type of the method is the same as the definition type of the attribute *Attn* because the intention of the getter method is to retrieve the value of that attribute. The procedures **getType**(*Pn,Cn, Attn, attribute, AttType*) is used to retrieve the definition type of the attribute under consideration.

- FGT 2 in the FGT-list is used to add a *read* relation between the created getter method as a source of the *relation* and the attribute *Attn* as a destination. The *read* relation between the two *objects* is an indication that the getter method has *read* access to the attribute *Attn.* This means that one or more statements in *Methn* will have a *read* access on the value of the attribute *Attn*. This will be reflected at the code-level.

- Precondition conjunct (1) is covered by precondition conjuncts of the FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjunct (2) is covered by precondition conjuncts of the FGT 2 in the FGT-list (*section 4.2.2.1.A*). There is no need to add precondition conjuncts at the refactoring-level.

Note that this refactoring indeed preserves system behaviour, but is matter futile if applied on its own. Normally, it will be applied in a context where *Attn* is being accessed directly, and there is a need to encapsulate it. To do this, several more primitive refactorings need to be applied. Chapter 6 provides an example of how such encapsulation may be achieved by the application of various primitive refactorings, which together may be viewed as an example of a composite refactoring called **encapsulateAttribute**.

5.2.6 addSetter(*AttributeName*)

Where *AttributeName* has the following format: *Pn.Cn.Attn*.

Description

The refactoring adds a setter method in the class *Pn.Cn*, the intention of this method is to be used to set the value of the attribute *Attn* that is defined in the class *Pn.Cn*. For that the setter method has a parameter whose definition type is the same as the definition type of the attribute *Attn*.

Figure 5.2 shows the effect of the refactoring addSetter when it is applied on the *private* attribute *A.x* using: addSetter(*A.x*) .

Precondition Conjuncts

(1) The signature of the setter method is distinct from those all methods declared already in the class *Pn.Cn* or any of its ancestors.

(2) The attribute *Attn* is declared in the class *Pn.Cn*.

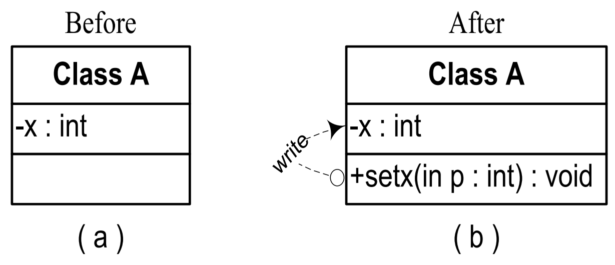


Figure 5.2: Class A before and after addSetter(*A.x*)

FGT-List

1. addObject(*Pn,Cn, Methn,_,_,type(basic,void,0),public,[(p,AttType)],method*)
2. addRelation(_,*Pn,Cn,Methn,_,[Tname],method,Pn,Cn,Attn,_,_,attribute,write*)

Note

- FGT 1 in the FGT-list is used to add the setter method, the name of the setter method *Methn* is formulated by using the procedure **concat**(*'set', Attn, Methn*) that is used to concatenate the word *'set'* with the attribute *Attn*. The return type of the method is *void* because the setter method returns no values. The setter method has only one parameter which has the same definition type as the definition type of the attribute *Attn* because the intention of the setter method is to set (change) the value of that attribute by using this parameter. The procedure **getType**(*Pn,Cn, Attn, AttType*) is used to retrieve the definition type of the attribute under consideration.

- FGT 2 in the FGT-list is used to add a *write* relation between the created setter method as a source of the *relation* and the attribute *Attn* as a destination. The *write* relation between the two *objects* is an indication that the setter method has a *write* access on the attribute *Attn*. The procedure **typeName**(*AttType, Tname*) is used to retrieve the type name (*int, float, …*) of the *AttType*. *Tname* is used in FGT 2 to specify the signature of the method *Methn*.

- Precondition conjunct (1) is covered by precondition of the FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjunct (2) is covered by precondition conjuncts of the FGT 2 in the FGT-list (*section 4.2.2.1.A*). There is no need to add precondition conjuncts at the refactoring-level.

5.3 Change Element Refactorings

These refactorings can be divided into two groups. The first group includes refactorings that are used to change the characteristics of the *object* elements in the system by changing the name, access mode and definition type of *object* elements. Note that one of the features of the proposed refactoring tool is that all the references to the *object* elements are done through the ID of the *object* elements and not through their names, so when we change the name of the *object* element, for example, then there is no need to change any references to that *object*.

The second group includes refactorings that are used to restructure *object* elements in the system by changing the hierarchal *relations* between *objects* or by moving *object* elements from one place to another or by redirecting member's accesses from one *object* element to another.

5.3.1 Changing Characteristics

5.3.1.1 renameClass(*ClassName, NewName*)

The refactoring changes the name of a class. (*For more details see Appendix B.2.1*)

5.3.1.2 renameMethod(*MethodName, MethTList, NewName*)

The refactoring changes the name of a method. (*For more details see Appendix B.2.2*)

5.3.1.3 renameAttribute(*AttributeName, NewName*)

The refactoring changes the name of an attribute. (*For more details see Appendix B.2.3*)

5.3.1.4 renameParameter(*ParameterName, MethTList, NewName*)

The refactoring changes the name of a parameter. (*For more details see Appendix B.2.4*)

5.3.1.5 changeClassAccess(*ClassName, NewAcces*)

The refactoring changes the access mode of a class. (*For more details see Appendix B.3.1*)

5.3.1.6 changeMethodAccess(*Methname, MethTList, NewAccess*)

The refactoring changes the access mode of a method. (*For more details see Appendix B.3.2*)

5.3.1.7 **changeAttributeAccess(***AttributeName, NewAccess***)**

The refactoring changes the access mode of an attribute. (*For more details see Appendix B.3.3*)

5.3.1.8 **changeMethodReturnType(***Methodname, MethTList, NewRType***)**

The refactoring changes the definition type of the return value of a method. (*For more details see Appendix B.3.4*)

5.3.1.9 **changeAttributeDefType(***AttributeName, NewDType***)**

The refactoring changes the definition type of an attribute. (*For more details see Appendix B.3.5*)

5.3.1.10 **changeParameterDefType(***Parametername, MethTList, NewDType***)**

The refactoring changes the definition type of a parameter. (*For more details see Appendix B.3.6*)

5.3.2 Change Structure (Restructuring)

5.3.2.1 **changeSuper(***ClassName, NewSuper***)**

Where

- *ClassName* has the following format: *Pn.Cn*
- *NewSuper* has the following format: *NewPn.NewCn*

Description

The refactoring changes the superclass of the class *Pn.Cn* to a new class *NewPn.NewCn***.**

Precondition Conjuncts

(1) Members of the old superclass or any of its ancestors are not referenced by instances of the class *Pn.Cn* or any of its descendents.

FGT-List

1. deleteRelation(_,*OldPn, OldCn,_,_,_,class, Pn,Cn, _,_,_,class, extends*)

2. addRelation(*isa, NewPn, NewCn,_,_,_, class, Pn,Cn, _,_,_,class, extends*)

Note

- In order to find the superclass of the class *Pn.Cn* we use the procedure **supClass**(*OldPn, OldCn, Pn,Cn*)

- FGT 1 is used to delete the *extends* relation between the old superclass *OldPn.OldCn* and the class *Pn.Cn*.

- FGT 2 is used to add the *extends* relation between the new superclass *NewPn.NewCn* and the class *Pn.Cn*.

- Note that even if class *Pn.Cn* or any of its descendants have a member *x* that is defined in the class *NewPn.NewCn* or any of its ancestor classes, adding the *extends* relation between the two classes will not cause a redefining of the member *x*. *Pn.Cn* and its descendants will still use their version of *x*. Thus, member *x* that is defined in the class *Pn.Cn* or one of its descendants is not affected by adding the *extends* relation.

- Also note that the new members that the class *Pn.Cn* and its descendants will inherit from the new superclass will not affect the behaviour of the system because these inherited members are not referenced by any instance or member of the class *Pn.Cn* or its descendants.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.2.3.B*). There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.2 moveMethod(*MethodName, NewClassName, MethTList*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *NewClassName* has the following format: *NPn.NCn*
- *MethTList* has the following format: *[*$Tname_1$*, *$Tname_2$*,…., *$Tname_n$*]*

Description

The refactoring moves a method from one class to another. The developer may need to do this when the two classes are highly coupled and the method to be moved *Methn* is extensively accessed members that are defined in the destination class. In this case, the developer concludes that the method is more related to the destination class and moving it will make the system more readable and simple. For example, in Figure 5.3(a), method *B.m* accesses the *private* attribute *A.x* through its getter and setter methods. The figure shows that method *B.m* does not access any members in the class *B*. It is reasonable to conclude that the method *B.m* is more related to the class *A* than class *B* and a developer might therefore prefer to move the method to the class A. For this, refactoring **moveMethod**(*B.m, A, [int]*) may be used. Note that in order to serve all the accesses (*calls*) to *B.m* from the other *object* elements in the system, the tool adds a method with the same signature in the source class *B*. Then a *call* relation is created between the two methods in the two classes. All the existing accesses to the method *B.m* will be now redirected to method *m* in its new location *A.m*.

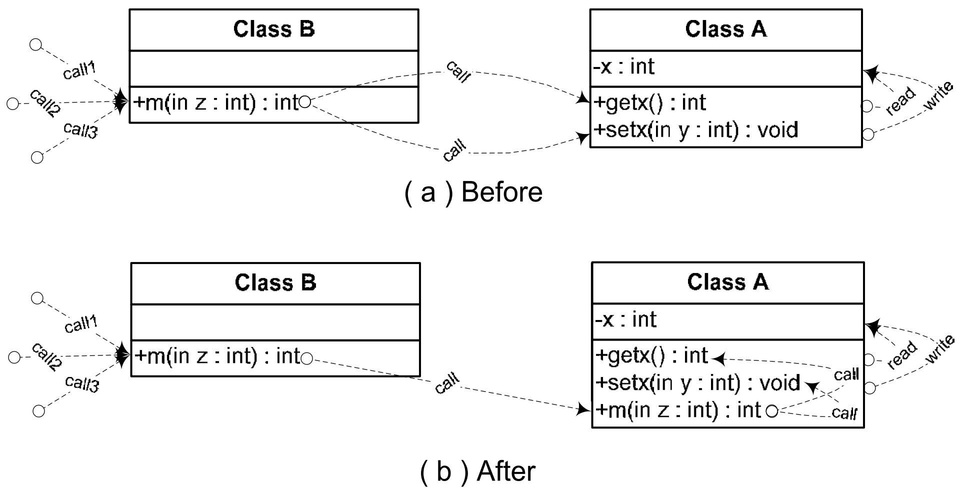


Figure 5.3: Class A & B before and after moveMethod(*B.m, A, [int]***)**

Precondition Conjuncts

(1) The signature of the method *Methn* is distinct from those all methods declared already in the class *NPn.NCn* or any of its ancestors.

FGT-List

1. **For** each relational element that the *Methn* is the source object of **do** {

```
      deleteRelation(_,Pn,Cn,Methn,_,PrmLT,method,$TPn_i$,$TCn_i$,$TMem_i$,$TPrm_i$, $TPrmLT_i$,
      $TOT_i$,$RelType_i$) }
2. addObject(NPn, NCn, Methn,_,_, RetDefT, public, MethTList, method)
3. For each relational element deleted in stage 1 do {
      addRelation(_,NPn,NCn,Methn,_,PrmLT,method,$TPn_i$,$TCn_i$,$TMem_i$,$TPrm_i$, $TPrmLT_i$,
      $TOT_i$,$RelType_i$) }
4. addRelation(_,Pn,Cn,Methn,_,PrmLT,method,NPn,NCn,Methn,_,PrmLT, method, call)
```

Note

- Stage 1 is used to delete all the *relational* elements that exist between the method *Methn* and any other *object* elements in the system, where the method *Methn* is the source of the *relation.* Thus, all *Methn* accesses to the other *objects* will be deleted. Note that this stage will generate a deleteRelation FGT for each existing *relation*. In Figure 5.3(a) the two *call* relations from the method *B.m* to the methods *A.gets* and *A.setx* will be deleted at this stage.

- FGT 2 is used to add a new method with the same signature as *Methn* to the destination class *NPn.NCn*. Figure 5.3(b) shows that the method *m* is added to the class *A*.

- All the *relational* elements that were deleted during stage 1 will be added by stage 3 with the newly created method in the destination class *NPn.NCn* as a source of these *relational* elements. Figure 5.3(b) shows that the two *call* relations that were deleted during stage 1 are added between the method *A.m* as a source and the two methods *A.setx* and *A.getx* as destinations.

- FGT 4 is used to create a *relational* element of type *call* between the method *Pn.Cn.Methn* and the new method *NPn.NCn.Methn*. The purpose of this *relation* is to forward all the accesses from all *object* elements to the method *Methn* in its old location to its new location. Figure 5.3(b) shows that a new *call* relation is created between the method *B.m* and the method *A.m*, so all the accesses to the method *B.m* is still valid and forwarded to the method *A.m*.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 2. There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.3 moveAttribute(*AttributeName, NewClassName*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *NewClassName* has the following format: *NPn.NCn*

Description

The refactoring moves an attribute from one class to another. This primitive refactoring is typically used if the attribute under consideration is intensively accessed—through its getter and setter methods—by *object* members defined in another class than by members defined in its own class. As shown in Figure 5.4(a) below, there are many accesses from methods in the class *B* to the attribute *A.x*. In this case it is preferred to move the attribute from class *A* to class *B*, and for this refactoring **moveAttribute**(*A.x, B*) is used.

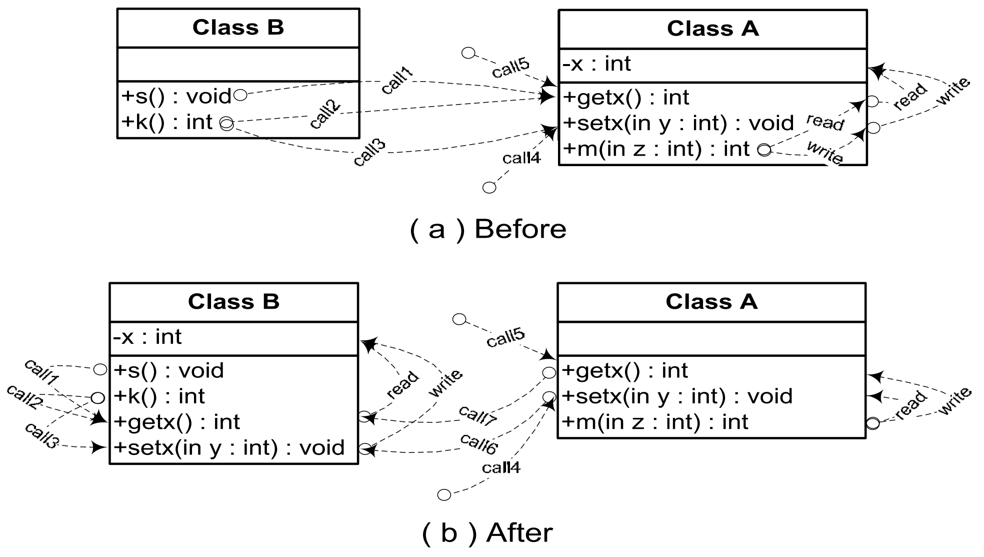


Figure 5.4: Class A & B before and after moveAttribute(*A.x, B*).

Precondition Conjuncts

(1) The attribute *Attn* is distinct from those all attributes that are declared already in the class *NPn.NCn* or any of its ancestors.

FGT-List

1. deleteRelation(_,*Pn,Cn, getMethn,_,[ ],method,Pn,Cn,Attn,_,_,attribute,read*)
2. deleteRelation(_,*Pn,Cn, setMethn,_,[Tname],method,Pn,Cn, Attn,_,_,attribute, write*)
3. deleteObject(*Pn,Cn, Attn,_,_,attribute*)
4. addObject(*NPn,NCn, Attn,_,_,AttType, private,_,attribute*)
5. **addGetter**(*NPn.NCn.Attn*)
6. **addSetter**(*Npn.NCn.Attn*)
7. addRelation(_,*Pn,Cn, getMethn,_,[ ],method,NPn,NCn, getMethn,_,[ ], method, call*)
8. addRelation(_,*Pn,Cn, setMethn,_,[Tname],method,NPn,NCn, setMethn,_, [Tname],method, call*)
9. deleteRelation(_,*NPn,NCn,* $NMem_i$,_,_,NOT_i,*Pn,Cn, getMethn,_,[ ],method,call*)
10. addRelation(_,*NPn,NCn,* $NMem_i$,_,_,NOT_i,*NPn,NCn, getMethn,_,[ ],method, call*)
11. deleteRelation(_,*NPn,NCn,* $NMem_i$,_,_,NOT_i,*Pn,Cn, setMethn,_,[Tname], method,call*)
12. addRelation(_,*NPn,NCn,* $NMem_i$,_,_,NOT_i,*NPn,NCn, setMethn,_,[Tname], method, call*)

Note

- FGTs 1 and 2 are used to delete the *read/write* relations in the source class *Pn.Cn* between the getter/setter methods and the attribute *Attn*. In Figure 5.4(a) the two *read/write* relations from the methods *A.getx*/*A.setx* to the attribute *A.x* will be deleted at this stage.

- FGT 3 is used to delete the attribute *Attn* from the source class *Pn.Cn*.

- FGT 4 is used to add the attribute *Attn* to the destination class *NPn.NCn*.

- Then in stages 5 and 6, the two refactorings **addGetter** (describe in section 5.1.5) and **addSetter** (describe in section 5.1.6) are used to create a getter and a setter methods for the attribute *NPn.NCn.Attn*. Figure 5.4(b) shows that *read/write* relations are created from the methods *B.getx/B.setx* to the attribute *B.x*.

- FGTs 7 and 8 are used to create a *call* relations between the getter/setter methods in the source and the destination classes. Figure 5.4(b) shows two *call* relations (*call6* and *call7*) are created from *A.getx/A.setx* to *B.getx/B.setx*.

- FGTs 9 to 12 are used to redirect all the *call* relations from the class *NPn.NCn* to the getter/setter methods in the class *Pn.Cn*. These calls are redirected to the new getter/setter methods in the class *NPn.NCn*. In Figure 5.4(b) the relations *call1, call2* and *call3* are redirected to the methods *B.getx/B.setx*

- Precondition conjunct (1) is covered by the set of precondition conjuncts of the FGT 4**.** There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.4 attributeReadsToMethodCall(*AttributeName, MethodName, MethTList*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname*$_1$*, Tname*$_2$*,…., Tname*$_n$*]*

Description

The refactoring redirects all *read* accesses to a specific attribute *Attn* to be through a getter method *Methn*. The method *Methn* will return the value of the attribute *Attn* to the calling *object* element**.**

Precondition Conjuncts

(1) The access mode of the method *Methn* is equal to or less restricted than the access mode of the attribute *Attn*. This ensures that all the *read* accesses to the attribute *Attn* will be within the access scope of the method.

(2) The method *Methn* acts as a getter method to the attribute *Attn*. This means that when the method *Methn* is called it will return the value of the *Attn* to the calling object.

FGT-List

For each relational element of type *read* with *Pn.Cn.Attn* as the destination object **do** {

1. deleteRelation(*_,SPn*$_i$*, SCn*$_i$*,SMethn*$_i$*,_,PrmLT*$_i$*,SOT*$_i$*, Pn,Cn, Attn,_,_, attribute, read*)
2. addRelation(*_,SPn*$_i$*, SCn*$_i$*,SMethn*$_i$*,_,PrmLT*$_i$*,SOT*$_i$*, Pn,Cn,Methn,_, MethTList, method,call*) **}**

Note

- The destination of all *relational* elements—whose destination *object* is the attribute *Pn.Cn.Attn* and whose type is *read*—will be changed to be *Pn.Cn.Methn* instead of *Pn.Cn.Attn*.

- Precondition conjunct (1) is covered by the precondition of FGT 2 because in order to create the *call* relation, the destination method *Pn.Cn.Methn* should be accessible. Precondition conjunct (2) is not covered by precondition conjuncts of the FGTs in the FGT-list because there is no guarantee that the method *Methn* acts as a getter method to the attribute *Attn*. In order for the method *Methn* to be a getter method of the attribute *Attn,* the return type of the *Methn* should be the same as the return type of the *Attn. In addition* there has to be a *read* relation between the *Methn* and the *Attn*. These precondition conjuncts will therefore need to be specified at the refactoring-level of this primitive refactoring.

5.3.2.5 attributeWritesToMethodCall(*AttributeName, MethodName, MethTList*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[*$Tname_1$*,* $Tname_2$*,…., *$Tname_n$*]*

Description

The refactoring redirects all the *write* accesses to a specific attribute *Attn* to be through a setter method *Methn*. The setter method *Methn* will receive a value from the calling *object* element and set the value of the attribute accordingly.

Precondition Conjuncts

(1) The access mode of the method *Methn* is equal to or less restricted than the access mode of the attribute *Attn.* This ensures that all the accesses to the attribute will be within the access scope of the method.

(2) The method *Methn* acts as a setter method to the attribute *Attn*. This means that when the method *Methn* is called it will receive a value of the same definition type as the *Attn* and the method will set the value of the *Attn* to this value.

FGT-List

For each relational elements of type *write* and *Pn.Cn.Attn* is the destination object **do** {

1. deleteRelation(*_,SPn_i,SCn_i,$SMehN_i$,_,$PrmLT_i$,SOT_i, Pn,Cn,Attn,_ ,_, attribute, write*)
2. addRelation(*_,SPn_i,SCn_i,$SMehN_i$,_,$PrmLT_i$,SOT_i, Pn,Cn,Methn,_,[Tname],method, call*)

   }

Note

- The destination of all *relational* elements—whose destination object is attribute *Pn.Cn.Attn* and whose *relation* type is *write*—will be changed to be *Pn.Cn.Methn* instead of *Pn.Cn.Attn*.

- Precondition conjunct (1) is covered by FGT 2 because in order to create the *call* relation, the destination method *Pn.Cn.Methn* should be accessible. Precondition conjunct (2) is not covered by precondition conjuncts of the FGTs in the FGT-list because even though the FGT 2 ensures that the *Methn* has a parameter of the same type as the attribute *Attn*, there is no guarantee that the method *Methn* has *write* access to the attribute *Attn*. For this we need to check if there is a *write* relation between the method *Methn* and the attribute *Attn*. This precondition conjunct will be defined at the refactoring-level of this primitive refactoring.

5.3.2.6 pullUpMethod(*SubClassesNames, Methn, MethTList*)

Where

- *SubClassesNames* has the following format: *[($SubPn_1$,$SubCn_1$), ($SubPn_2$, $SubCn_2$),.., ($SubPn_n$, $SubCn_n$)]* where items in the list represent the names of the subclasses that the refactoring will pull the method from.
- *MethTList* has the following format: *[$Tname_1$, $Tname_2$,…., $Tname_n$]*

Description

The refactoring pulls up a method *Methn* from a list of subclasses *SubClassesNames* to their common superclass. If all subclasses in list have the same method with the same signature and the same effect. Then inconsistencies caused by not changing all these methods equally can be avoided by pulling up this method to their common superclass. It is clear that pulling the method up will not affect the behaviour of the system because all the subclasses after refactoring will have this method by inheritance.

The access mode of the method *Methn* should not be more general than the access modes of the corresponding versions of the method in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

Precondition Conjuncts

(1) The method *Methn* should not be declared in the superclass nor in any of its ancestors.

(2) The access mode of the method *Methn* in the subclasses is not *private*.

(3) All the references made by *Methn* to the other *object* elements should be visible from the superclass.

(4) The signature of *Methn* in all the subclasses in the list *SubClassesNames* should be the same.

Note 1: Precondition conjunct (4) is not necessarily sufficient to legitimate a pull up refactoring. In addition in should be the case that the postcondition conjuncts of the various methods in the subclasses are compatible. Technically, one might say that the postcondition of at least one method should logically entail the postcondition conjuncts of all the others. In this case, the method with the strictest postcondition should be pulled up. (Further explanation of this point is beyond the scope of this thesis.)

However, the prototype refactoring tool built for the purposes of this thesis does not require that postcondition information should be available. It merely considers information embedded in UML class diagrams as well as some limited information embedded in the code. It is therefore the responsibility of the tool user to ascertain the compatibility of method postcondition conjuncts before carrying out a pull up refactoring. The tool will, however, check compliance with precondition four as an approximation of the more rigorous requirement of postcondition compatibility.

FGT-List

1. addObject(*SupPn,SupCn,Methn,_,_,MethRType,OAMode,MethTList, method*)
2. **For** each subclass in the *SubClassesNames* list **do** {

deleteObject(*SubPn$_i$*,*SubCn$_i$*, *Methn*,_, *MethTList*, *method*) }

Note

- FGT 1 is used to add the method *Methn* in the superclass with the same signature as in the subclasses. The method access mode *OAMode* is calculated according to the rule mentioned above. For this, the procedure **objectAMode**(*SubPn$_i$*, *SubCn$_i$*, *Methn*, *MethTList*, *method*, *SubOAmode*) is used.

- In stage 2, method *Methn* will be deleted from each one of the subclasses that is found in the list *SubClassesNames*.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjuncts (2) and (3) are not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as refactoring-level precondition conjuncts for this refactoring. Precondition conjunct (4) is covered by FGTs in stage 2.

5.3.2.7 pushDownMethod(*SuperClassName*, *MethodName*, *MethTList*)

Where

- *SuperClassName* has the following format: *SupPn.SupCn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname$_1$, Tname$_2$,…., Tname$_n$]*

Description

The refactoring pushes down a method *Methn* from a superclass to all its subclasses. This refactoring can be used if the *Methn* is not referenced in some of the subclasses. In such a case, this refactoring is used to push down the method to all the subclasses. It is thereafter deleted from those subclasses where it is not referenced, using the **deleteMethod** refactoring. The access mode of *Methn* in all the subclasses will be the same as its access mode in the superclass.

Precondition Conjuncts

(1) The method *Methn* should not be declared in any of the subclasses of the superclass.

(2) The method *Methn* should not be referenced by members of the superclass, since it will be deleted from the superclass and these referenced will be not defined anymore.

(3) The method *Methn* should not access any of the *private* members of the superclass.

(4) The access mode of the method *Methn* in the superclass should not be *private*.

FGT-List

1. **For** each subclass of the class *SuperClassName* **do** {

    addObject(*SubPn$_i$, SubCn$_i$, Methn,_,_,MethType, OAMode, MethTList, method*) }

2. deleteObject(*SupPn, SupCn, Methn, _, MethTList, method*)

Note

- Stage 1 is used to add the method *Methn* in all the subclasses of the class *SupClassName*. The signature of the method will be the same as defined in the superclass.

- FGT 2 is used to delete the method *Methn* from the superclass.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjuncts (2), (3) and (4) are not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as refactoring-level precondition conjuncts.

5.3.2.8 pullUpAttribute(*SupClassName, Attn*)

Where *SuperClassName* has the following format: *SupPn.SupCn*

Description

The refactoring pulls up an attribute *Attn* to the superclass *SupClassName* from all subclasses where it is defined. If the access mode of the attribute *Attn* where it is currently defined is *public* then it will be *public* in the superclass class; otherwise, the access mode of the *Attn* in the superclass will be *protected*. None of the references to the attribute in the subclasses and their descendants will be affected because they will inherit the attribute from the superclass. The refactoring is thus behaviour-preserving.

Precondition Conjuncts

(1) The attribute that to be pulled up *Attn* should not be declared in the superclass or one of its ancestors.

(2) The attribute *Attn* should be declared identically (have the same definition type) in all the subclasses where it is defined.

(3) The access mode of the attribute *Attn* in the subclasses may not be *private.*

FGT-List

```
1. addObject(SupPn,SupCn,Attn,_,_, AttType, OAMode,_,attribute)
2. For each subclass of SupPn.SupCn where Attn is defined do {
       deleteObject(SubPn_i,SubCn_i, Attn,_,_, attribute) }
```

Note

- FGT 1 is used to add the attribute *Attn* into the superclass. The definition type of the attribute will be found by the procedures **getType**(…). The attribute access mode *OAMode* is calculated according to the rule mentioned above. For this, the procedure **objectAMode**(*SubPn$_i$, SubCn$_i$, Attn, attribute,SubOAmode*) is used.

- In stage 2, the attribute *Attn* will be deleted from each one of the subclasses in which it is defined.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.C*) . Precondition conjunct (2) is not covered by precondition conjuncts of FGTs in the FGT-list. It requires that *Attn* be defined identically in all the subclasses. For checking this, the procedure **checkIdentically**(*Attn, ClassList, Identical*) may be used. The procedure takes as input the name of the attribute and the list of all subclasses where the attribute is defined. It then return true by the parameter *Identical* if the attribute *Attn* has the same definition type in all the classes in the list *ClassList*. Precondition conjunct (3) is not covered by precondition conjuncts of FGTs in the FGT-list. Precondition conjuncts (2) and (3) should be defined as refactoring-level precondition conjuncts.

5.3.2.9 pushDownAttribute(*SupClassName, AttributeName*)

Where

- *SuperClassName* has the following format: *SupPn.SupCn*
- *AttributeName* has the following format: *Pn.Cn.Attn*

Description

The refactoring pushes down an attribute *Attn* from a superclass to all its subclasses. This refactoring is useful if the *Attn* is not referenced in some of the subclasses. In such a case, this refactoring is used to push down the attribute to all the subclasses. It is thereafter deleted from those subclasses where it is not referenced, using the **deleteAttribute** refactoring. The access mode of *Attn* in all the subclasses will be the same as its access mode in the superclass.

Precondition Conjuncts

(1) The attribute *Attn* should be not referenced by members or instances of the superclass.

(2) The access mode of the attribute *Attn* in the superclass should not be *private*.

FGT-List

```
1. For each subclass of the SupClassName do {
       addObject(SubPn_i, SubCn_i, Attn,_,_, AttType, OAmode,_,attribute) }
2. deleteObject(SupPn, SupCn, Attn,_,_, attribute)
```

Note

- In stage 1 the attribute *Attn* will be added to all the subclasses of the class *SupClassName*. The definition type and access mode of the attribute will be the same as in the superclass.

- FGT 2 is used to delete the attribute *Attn* from the superclass.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 2 in the FGT-list (*section 4.2.1.1.C*). Precondition conjunct (2) is not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as a refactoring-level precondition.

5.4 Delete Element Refactorings

These refactorings are used to delete unreferenced *object* elements from system.

5.4.1 deleteClass(*ClassName*)

Where *ClassName* has the following format: *Pn.Cn*

Description

The refactoring deletes unreferenced class *Cn* from the package *Pn*. For this refactoring the class *Pn.Cn* may have a superclass but it should not have any subclasses.

Precondition Conjuncts

(1) The class *Pn.Cn* should not be referenced by any other *object* elements outside the class.

(2) The class *Pn.Cn* has no subclasses.

FGT-List

1. **If** the class to be deleted has a superclass **then**

   deleteRelation(_,*SupPn,SupCn*, _,_,_,_,*Pn,Cn*,_,_ _,_,*extends*)

2. Delete all the relational elements between any two object elements defined in the class *Pn.Cn*

   deleteRelation(_,*Pn,Cn*, _,_,_,_,*Pn,Cn*,_,_ _,_,_)

3. Delete all the methods defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn*, $Methn_i$,_,_ *method*)

4. Delete all the attributes defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn*, $Attn_i$,_,_,*attribute*)

5. deleteObject(*Pn, Cn*,_,_,_ *class*)

Note

- In order to delete a class by using the FGT deleteObject, the class should be empty (no members) and should also stand alone (no superclasses or subclasses). For the refactoring **deleteClass**, one of the precondition conjuncts is that the class should not have subclasses, although it may have a superclass and it may also have members defined in it. Therefore, to delete the class *Pn.Cn*, the tool should first check if there is a superclass for the class *Pn.Cn* by using the procedure **supClass**(*SupPn,SupCn,Pn,Cn*); if there is then FGT 1 is used to delete the *extends* relation between the classes *SupPn.SupCn* and *Pn.Cn*.

- Although the members of the class *Pn.Cn* are unreferenced by any *object* elements defined outside the class (this is ensured by one of the refactoring precondition conjuncts), references between the different object elements in the class *Pn.Cn* may exist. All these references have to be deleted. Stage 2 in the FGT-list is used for this purpose.

- In stage 3 and 4, all members of the class *Pn.Cn* are deleted.

- FGT 5 is used to delete the class *Pn.Cn*.

- Precondition conjunct (1) is covered by the set of precondition conjuncts of FGTs 3, 4 and 5 of the FGT-list (*section 4.2.1.5*). Precondition conjunct (2) is covered by the set of precondition conjuncts of the FGT 5 because the procedure **isReferenced**(...) will also check if there is any *extends* relation with the class *Pn.Cn*. There is therefore no need to add precondition conjuncts at the refactoring-level for this refactoring.

5.4.2 deleteMethod(*MethodName, MethTList*)

The refactoring deletes an unreferenced method from specific class. (*For more details see Appendix B.4.1*)

5.4.3 deleteAttribute(*AttributeName*)

The refactoring deletes an unreferenced attribute from specific class. (*For more details see Appendix B.4.2*)

5.4.4 deleteParameter(*Prmname, MethTList*)

The refactoring deletes a parameter from the parameter's list of specific method *Methn*. This refactoring is beneficial when, for example, a method's purpose is changed and there is a need to remove (and perhaps later add) parameters from (to) the method. (*For more details see Appendix B.4.3*)

5.5 Reflection on this Chapter

Note that for each primitive refactoring presented in this chapter, there is a corresponding procedure in the prototype tool that receives a set of parameters as input and generates an FGT-list. It should be emphasized that the list of generated FGTs is dependent on the system. For example, the **deleteClass** primitive refactoring (which deletes an unreferenced class from the system) has a corresponding procedure that builds a *system-dependent* FGT-list by carrying out the following steps:

1. **If** the class to be deleted has a superclass **then**

   deleteRelation(_,*SupPn,SupCn,* _,_,_,_,*Pn,Cn*,_,_, _,_,*extends*)

2. Delete all the relational elements between any two object elements defined in the class *Pn.Cn*

   deleteRelation(_,*Pn,Cn,* _,_,_,_,*Pn,Cn*,_,_, _,_,_)

3. Delete all the methods defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn,* $Methn_i$,_,_, *method*)

4. Delete all the attributes defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn,* $Attn_i$,_,_,*attribute*)

5. deleteObject(*Pn, Cn*,_,_,_, *class*)

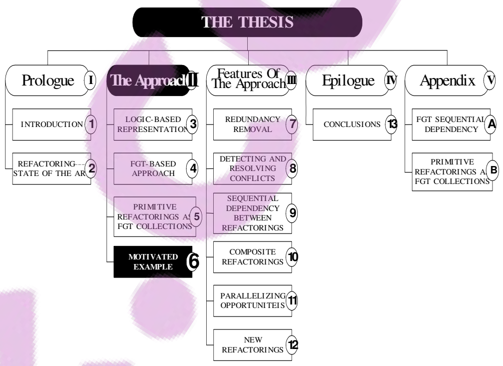
After executing this procedure the list of FGTs is returned. As a second stage a set of FGT-DAGs corresponding to this FGT-list may be generated, as discussed in chapter 4. Recall that there may be one or more FGT-DAGs for that primitive, depending on the sequential dependencies between FGTs in the produced list.

In later chapters, it will be seen that various operations can be performed on arbitrary FGT-DAG sets. For example, it might be possible to reduce them (see chapter 7), to detect conflicts in them (see chapter 8), or to establish sequential dependencies between arbitrary FGT-DAGs (see chapter 9), or representing different composite refactorings from primitive ones (see chapter 10), perhaps process them in parallel (see chapter 11). However, before proceeding to these matters, a motivating example is provided in the next chapter.

CHAPTER 6

MOTIVATED EXAMPLE

6.1 LAN Simulation


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

To illustrate the approach outlined above, an example is presented that is frequently used for teaching refactoring: the simulation of a Local Area Network (LAN) [13]. Initially there are five classes: *Packet*, *Node* and the three subclasses: *Workstation*, *PrintServer* and *FileServer*. The idea is that all *Node* objects are linked to each other in a token ring network (via the *NextNode* variable) and that they can send or accept a *Packet* object. *PrintServer*, *FileServer* and *Workstation* refine the behaviour of *Node* objects. A *Packet* object can only originate from a *Workstation* object, and sequentially visits every *Node* object in the network until it reaches its receiver that accepts the *Packet,* or until it returns to its originator *Workstation* object (indicating that the *Packet* cannot be delivered).

The UML class diagram for the LAN example is shown in Figure 6.1. The dashed arrows represent the extra information extracted from the code-level of the LAN system which is shown in Figure 6.2. Recall that in the approach the interest of the code-level is limited to the *access-related* information that exists between the different object elements in the class diagram.

Suppose that it is required to enhance the structure of the LAN model as follows:

1. Encapsulate the attribute *originator* in the *Packet* class. This refactoring is useful for increasing modularity, by avoiding direct accesses of the local state of a packet. For this restructuring, the composite refactoring **encapsulateAttribute** will be used.

2. As another enhancement, it has been decided to create a new class *Server* to be a superclass of the *PrintServer*, *FileServer* classes and subclass of the *Node* class. The purpose of this refactoring is to show that the classes *PrintServer* and *FileServer* are similar in nature.

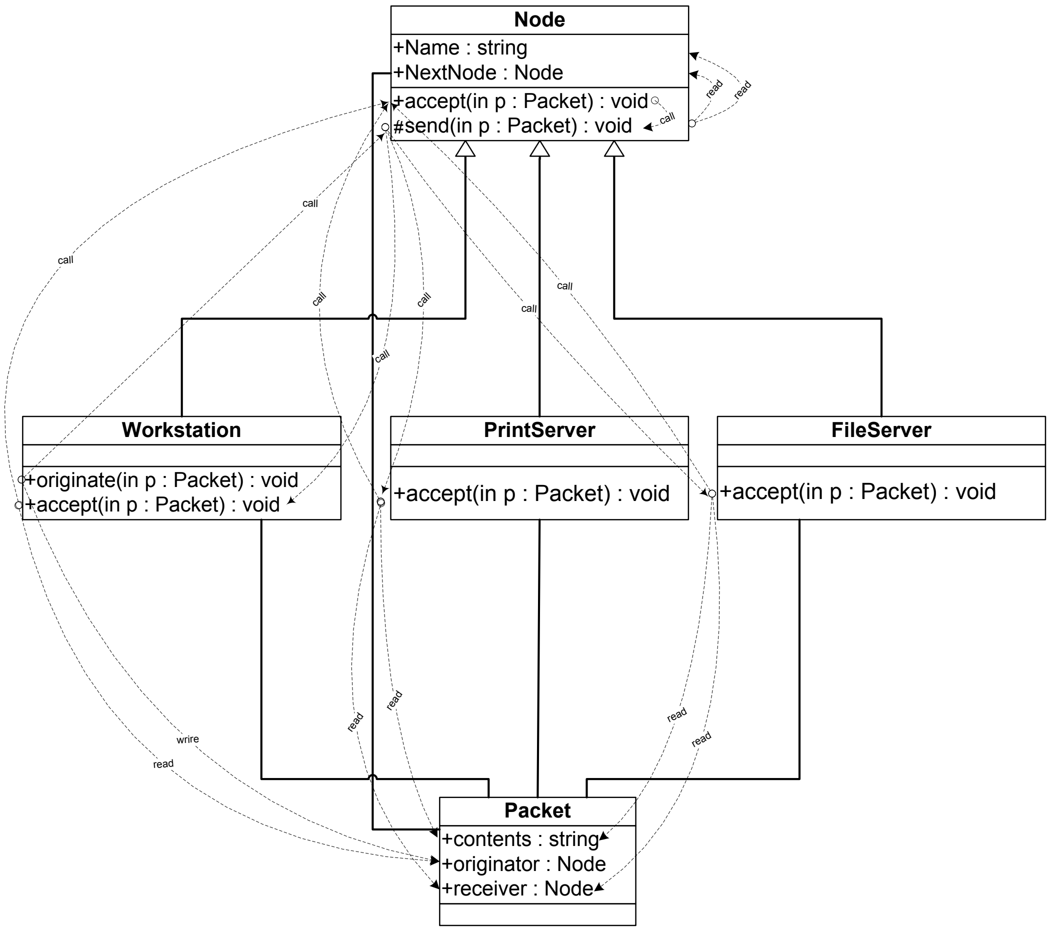

Node
+Name : string
+NextNode : Node
+accept(in p : Packet) : void
#send(in p : Packet) : void
Workstation
+originate(in p : Packet) : void
+accept(in p : Packet) : void
PrintServer
+accept(in p : Packet) : void
FileServer
+accept(in p : Packet) : void
Packet
+contents : string
+originator : Node
+receiver : Node
call
read
wrire

Figure 6.1: A UML class diagram of the LAN simulation before refactoring

```
package Lan;
class Node {
 public String Name;
 public Node NextNode;
 public void accept(Packet p) {
 this.send(p); }
 protected void send(Packet p) {
 System.out.println(Name - NextNode.Name);
 this.NextNode.accept(p); }
}
class Packet {
 public String contents;
 public Node originator;
 public Node receiver;
}
class PrintServer extends Node {
 public void accept(Packet p) {
 if(p.receiver == this)
   System.out.println(p.contents);
 else super.accept(p); }
}
class FileServer extends Node {
 public void accept(Packet p) {
 if(p.receiver == this)
   System.out.println(p.contents);
 else super.accept(p); }
}
class Workstation extends Node {
 public void originate(Packet p) {
 p.originator = this;
 this.send(p); }
 public void accept(Packet p) {
 if(p.originator == this)
   System.err.println("no destination");
 else super.accept(p); }}
```

Figure 6.2: A code-level implementation of the LAN simulation before refactoring

They can both accept a packet sent by another node in the network and process it in the same way. For this restructuring, the composite refactoring **createClass** will be used.

3. Thereafter, it is intended to pull up the method *accept* from *FileServer*, *PrintServer* classes to the class *Server* that was created in the previous stage. For this restructuring, the primitive refactoring **pullUpMethod** will be used.

The following sections show how the system is represented as a set of logic-terms; and how each of the above composite refactorings can be seen as a sequence of primitive refactorings, each of which can be represented as an FGT-list. Each composite refactoring therefore has a corresponding FGT-list associated with it. The chapter does not focus on mapping these refactoring's FGT-lists to FGT-DAG sets, since the forthcoming chapters will pay considerable attention to FGT-DAG sets. Here, instead, the FGT-lists are assumed to transform the original system to the refactored one. A number of subtleties relating to system representation are pointed out.

It should be noted that because the **encapsulateAttribute** and **createClass** composite refactorings could occur quite commonly, they have been implemented as procedures in the prototype tool. Sections 6.3 and 6.4 will show how they are to be invoked.

6.2 Logic-Based Representation

Before doing any refactoring, the class diagram and the extra information extracted from the code-level should be represented as collection of logic-terms as discussed in chapter 3. Figure 6.3 shows the collection of logic-terms for the LAN simulation example. All refactorings will be done on this underlying representation of the model.

```
package(0,00,Lan,[1,2,3,4,5]).
class(1,0,Node,public,[1001,1002],[10001,10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002, 1, accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,3002).
call(1000003,_,1001,4002).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000010,isa,1,3).
extends(10000011,isa,1,4).
extends(10000012,isa,1,5).
class(2,0,Packet,public,[],[20001,20002,20003]).
attribute(20001,2,contents,type(basic,string,0),public).
attribute(20002,2,originator,type(complex,1,0),public).
attribute(20003,2,receiver,type(complex,1,0),public).
```

```
class(3,0,FileServer,public,[3002],[]).
method(3002,3,accept,type(basic,void,0),public,[300002]).
parameter(300002,3002,p,type(complex,2,0)).
read(3000001,_,3002,20001).
read(3000003,_,3002,20003).
call(3000005,_,3002,1002).
class(4,0,PrintServer,public,[4002],[]).
method(4002,4,accept,type(basic,void,0),public,[400002]).
parameter( 400002,4002,p,type(complex,2,0)).
read(4000001,_,4002,20001).
read(4000003,_,4002,20003).
call(4000005,_,4002,1002).
class(5,0,Workstation, public,[5001,5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
write(5000001,_,5001,20002).
call(5000002,_,5001,1001).
read(5000004,_,5002,20002).
call(5000005,_,5002,1002).
```

Figure 6.3: Underlying logic representations of the LAN simulation before refactoring

6.3 encapsulateAttribute Refactoring

The refactoring **encapsulateAttribute** is a composite refactoring that is used to avoid direct access to a specific attribute. It was briefly mentioned in section 5.2.5. It includes the following actions:

1. Add getter and setter methods. This is done by using the primitive refactorings **addGetter** (*section 5.2.5*) and **addSetter** (*section 5.2.6*).

2. Replace accesses to the attribute by calls to the newly created methods. This is done by using the primitive refactorings **attributeReadsToMethodCall** (*section 5.3.2.4*) and **attributeWritesToMethodCall** (*section 5.3.2.5*) primitive refactorings.

3. Make the attribute *private*. This is done by using the primitive refactoring **changeAttributeAccess** (*section 5.3.1.7*).

To encapsulate the attribute *Packet.originator* we call the **encapsulateAttribute** procedure:

encapsulateAttribute(*'Lan','Packet',originator*)

The function will produce a collection of FGTs which represent the transformation actions that are needed to perform the encapsulation of the attribute as shown in the right column of Table 6.1. The collection of primitive refactorings used in the function is shown in the middle column of the Table.

For example, in the primitive refactoring **attributeReadsToMethodCall** that has the following format:

attributeReadsToMethodCall($Dest_x$, $Dest_y$)

any *read* access from anywhere in the system to the destination $Dest_x$ will be redirected to a new destination $Dest_y$. This means that for each *read* access, two FGT operations will be produced, one to delete the original *read* access "*read relation*" from the source *S* to the destination $Dest_x$, this is done by FGT:

deleteRelation(*_, S,* $Dest_x$*, read*)

and the other to add a new *read* access from the source *S* to the new distention $Dest_y$, this is done by FGT:

addRelation(*_, S,* $Dest_y$*, read*).

In the LAN example, there is one *read* access from *Workstation.accept* method to the *Packet.originator* attribute. Accordingly, the primitive refactoring

attributeReadsToMethodCall(*'Lan','Packet',originator,'Lan','Packet', getoriginator,[]*)

will produce two FGTs:

deleteRelation(*_,Lan,Workstation,accept,_,[Packet],method,Lan,Packet,originator,_,_, attribute,read*)

addRelation(*_,Lan,Workstation,accept,_,[Packet],method,Lan,Packet,getoriginator,_,[], method,call*)

Table 6.1: encapsulateAttribute refactoring

Com-posite Ref.	Seq. Of Primitive Refactorings	Seq. Of FGTs For Each Primitive Refactoring
encapsulateAttribute(*'Lan', 'Packet', originator*)	addGetter(*'Lan', 'Packet', originator*)	**FGT1:** addObject(*Lan,Packet,getoriginator,_,_,type(complex, Node,0), public,[ ],method*) **FGT2:** addRelation(_,*Lan,Packet,getoriginator,_,[ ],method, Lan, Packet,originator,_,_,attribute,read*)
	addSetter(*'Lan', 'Packet', originator*)	**FGT3:** addObject(*Lan,Packet,setoriginator,_,_,type(basic, void,0), public,[(p, type(basic,Node,0))],method*) **FGT4:** addRelation(_,*Lan,Packet,setoriginator,_,[Node], method, Lan,Packet,originator,_,_,attribute,write*)
	attributeReadsToMethodCall(*'Lan','Packet',originator, 'Lan','Packet',getoriginator, [ ]*)	**FGT5:** deleteRelation(_,*Lan,Workstation,accept,_,[Packet], method, Lan,Packet,originator,_,_,attribute,read*) **FGT6:** addRelation(_,*Lan,Workstation,accept,_,[Packet], method, Lan,Packet,getoriginator,_,[ ],method,call*)
	attributeWritesToMethodCall(*'Lan','Packet',originator, 'Lan','Packet',setoriginator, ['Node']*)	**FGT7:** deleteRelation(_,*Lan,Workstation,originate,_,[Packet], method,Lan,Packet,originator,_,_,attribute,write*) **FGT8:** addRelation(_,*Lan,Workstation,originate,_,[Packet], method ,Lan,Packet,setoriginator ,_,[Node], method,write*)
	changeAttributeAccess(*'Lan', 'Packet',originator,private*)	**FGT9:** changeOAMode(*Lan,Packet,originator,_,_,attribute, public, private*)

When the tool applies the nine FGTs that are produced/extracted from the composite refactoring **encapsulateAttribute** on the LAN system, the representation of the *Packet* and *Workstation* classes will be affected. Figure 6.4 shows the underlying logic representation of the two classes before and after applying the refactoring. The figure also shows the ID of each FGT alongside each logic-terms that is affected by it.

```
class(2,0,Packet,public,[],[20001,20002,20003]).
attribute(20001,2,contents,type(basic,string,0),public).
attribute(20002,2,originator,type(complex,1,0),public).
attribute(20003,2,receiver,type(complex,1,0),public).
class(5,0,Workstation, public,[5001,5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
write(5000001,_,5001,20002).                              FGT7
call(5000002,_,5001,1001).
read(5000004,_,5002,20002).                               FGT5
call(5000005,_,5002,1002).
```

(a)

```
class(2,0,Packet,public,[-17, -15],[20001, 20002, 20003]).
method(-15,2,getoriginator,type(complex,1,0),public,[]).      FGT1
method(-17,2,setoriginator,type(basic,void,0),public,[-18]).  FGT3
attribute(20001,2,contents,type(basic,string,0),public).
attribute(20002,2,originator,type(complex,1,0),private).      FGT9
attribute(20003,2,receiver,type(complex,1,0),public).
parameter(-18,-17,p,type(complex,1,0)).                       FGT3
read(-16,_,-15,20002).                                        FGT2
write(-19,_,-17,20002).                                       FGT4
class(5,0,Workstation,public,[5001, 5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
call(-20,_,5001,-17).                                         FGT8
call(5000002,_,5001,1001).
call(-21,_,5002,-15).                                         FGT6
call(5000005,_,5002,1002).
```

(b)

Figure 6.4: Packet & Workstation classes before and after encapsulateAttribute refactoring

6.4 createClass Refactoring

The refactoring **createClass** is a composite refactoring that is used to create a new class. The new class may be a standalone class or a super/sub (or both) of other classes, depending on the parameters that are used in the refactoring. It includes the following actions:

1. Add a new class. This is done by using the primitive refactoring **addClass** (*section 5.2.1*).

2. Change the superclass of the specific class from one class to another. This is done by using the primitive refactoring **changeSuper** (*section 5.3.2.1*).

In the motivated example, to create the class *Server* the composite refactoring procedure **createClass** is invoked:

$$\text{createClass}(\ \underbrace{\textit{'Lan', 'Server', public,}}_{\textit{The New Class}}\ \underbrace{\textit{'Lan', 'Node',}}_{\textit{Superclass}}\ \underbrace{\textit{['Lan', 'FileServer', 'Lan', 'PrintServer']}}_{\textit{List Of Subclasses}}\ )$$

As indicated in the list of parameters, the refactoring will create a new class, *Lan.Server*, with access mode *public*. The new class will be subclass of the class *Lan.Node* and superclass of the classes *Lan.FileServer* and *Lan.PrintServer*. Note that the subclasses are included in a list which can have as many subclasses as required. If the list is empty this means that the new class will not have any subclasses. The same also for the superclass parameter: if it is null then the new class will not have a superclass. The middle column of Table 6.2 shows the list of primitive refactorings that are used to construct the **createClass** refactoring, while the right column shows the collection of FGTs that are produced for each refactoring.

Table 6.2: createClass refactoring

Com-posite Ref.	**Seq. Of Primitive Refactorings**	**Collection Of FGTs**
createClass(*'Lan','Server',public,'Lan','Node', ['Lan','FileServer','Lan','PrintServer'])*	addClass(*'Lan','Server',public*)	**FGT1:** addObject(*Lan,Server,_,_,_,public,_, class*)
	changeSuper(*'Lan','Server','Lan','Node'*)	**FGT2:** addRelation(*isa,Lan,Node,_,_,_,class, Lan, Server, _,_,_,class,extends*)
	changeSuper(*'Lan','FileServer', 'Lan','Server'*)	**FGT3:** deleteRelation(*_,Lan,Node,_,_,_,class, Lan, FileServer, _,_,_,class,extends*) **FGT4:** addRelation(*isa,Lan,Server,_,_,_,class, Lan, FileServer,_,_,_,class,extends*)
	changeSuper(*'Lan','PrintServer', 'Lan','Server'*)	**FGT5:** deleteRelation(*_,Lan,Node,_,_,_,class, Lan, PrintServer,_,_,_,class,extends*) **FGT6:** addRelation(*isa,Lan,Server,_,_,_,class, Lan, PrintServer,_,_,_,class,extends*)

6.5 pullUpMethod Refactoring

The refactoring **pullUpMethod** is a primitive refactorings that is used to pull up a method from a list of subclasses to a superclass. For more details return to section *5.3.2.6*.

In the motivated example, to pull up the method *accept* from *FileServer*, *PrintServer* classes to the class *Server* the procedure

pullUpMethod(*['Lan','FileServer','Lan','PrintServer'],accept,['Packet']*) is called. The parameters show that the subclasses from which to pull up the method are inserted in a list. Thus, as many subclasses as desired can be given. The procedure will produce a collection of FGTs as show in the right column of Table 6.3.

Table 6.3: pullUpMethod refactoring

Primitive Ref.	**Collection Of FGTs**
pullUpMethod(*['Lan','FileServer', 'Lan','PrintServer'],accept,['Packet']*)	**FGT1**: addObject(*Lan,Server,accept,_,_,type(basic,void,0),public,[(p, type(complex, Packet,0))],method*) **FGT2**: deleteObject(*Lan,FileServer,accept,_,[Packet],method*) **FGT3**: deleteObject(*Lan,PrintServer,accept,_,[Packet],method*)

6.6 LAN after Refactorings

Figure 6.5 shows the produced collection of logic-terms for the LAN motivated example after applying the three refactorings **encapsulateAttribute**, **createClass** and **pullUpMethod**. Figure 6.6 shows the resulting UML class diagram based on the refactored version of the logic-terms. Figure 6.7 shows the modified code-level implementation of the UML class diagram after refactoring. Note that, in principle, the process of modifying the code from the refactored UML class diagram can be automated. However, details have not been investigated in this research.

The reader's attention is drawn to the reasons for colour-coding various entries in Figure 6.5. At first sight, it might appear strange that the **pullUpMethod** refactoring retained the *call*- and *read* relationships between the *Node* class on the one hand and the *FileServer* and *PrintServer* classes on the other. One might have expected that these should be relocated in the UML diagram (and logic-based representation thereof) to the new superclass in which the *accept* method has now been physically located.

However, these relationships refer to code-level activity. Despite the **pullUpMethod** refactoring, at the code-level the *call*- and *read* relationships have not been changed. All that has happened is that a *call* to method physically present in a class has become a *call* to an inherited method. This has happened, even though the actual code has not changed.

In terms of the visual representation of the UML class diagram that has been augmented with relationship information, all that needs to change is that the pulled up method should be shown in the superclass, and removed from the subclasses. However, Figure 6.6 shows the *accept* method in subclasses in blue, and provides a special note to indicate that this is for illustrative purposes, and by way of exception.

Notwithstanding these observations, the concrete representations of the *call*- and *read* relationships in the logic database have to be modified. This is because the *accept* method to which they refer is no longer in the respective subclasses, but in the superclass. To this end, at the logical level, a representation of the inherited method is retained for each inheriting subclasses. All relationship information is specified in terms of this representation.

Thus, in the specific example given, the *accept* method is represented in Figure 6.5 by three different entries. The first is a normal method with ID 57 in the class with ID 53. However, the method is also represented as an inherited method in the two subclasses. In these cases, special IDs are used for these two representations, namely 53_90, and 53_91 respectively. The 53 references the class ID in which the inherited method is to be found.

Similarly, the *read*- and *call* information has to be changed to reflect these new IDs. In Figure 6.5, all relations changed that relate to *FileServer* are given in blue, and those relating to *PrintServer* are given in red. The new method is given in green.

The reason for retaining this information is clear: it may be needed for a future refactorings, involving, for example, the **deleteMethod** refactoring. Recall that the precondition conjuncts for such a refactoring require that there should be no reference to the method to be deleted. The information reflected in Figure 6.5 will ensure that such precondition conjuncts may be properly checked.

```
package(0,00,Lan,[53, 1, 2, 3, 4, 5]).
class(1,0,Node,public,[1001, 1002],[10001, 10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002,1,accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,53_90).
call(1000003,_,1001,53_91).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000012,isa,1,5).
extends(54,isa,1,53).
class(2,0,Packet,public,[48, 46],[20001,20002,20003]).
method(46,2,getoriginator,type(complex,1,0),public,[]).
method(48,2,setoriginator,type(basic,void,0),public,[49]).
attribute(20001,2,contents,string,1,public).
attribute(20003,2,receiver,type(complex,1,0),public).
attribute(20002,2,originator,type(complex,1,0),private).
parameter(49,48,p,type(complex,1,0)).
read(47,_,46,20002).
write(50,_,48,20002).
```

```
class(3,0,FileServer,public,[53_90],[]).
method(53_90,3,accept,type(basic,void,0),public,[61]).
read(3000001,_,53_90,20001).
call(3000003,_,53_90,1002).
read(3000004,_,53_90,20003).
class(4,0,PrintServer,public,[53_91],[]).
method(53_91,4,accept,type(basic,void,0),public,[61]).
read(4000001,_,53_91,20001).
call(4000003,_,53_91,1002).
read(4000004,_,53_91,20003).
class(53,0,Server,public,[57],[]).
method(57,53,accept,type(basic,void,0),public,[61]).
parameter(61,57,p,type(complex,2,0)).
extends(55,isa,53,3).
extends(56,isa,53,4).
class(5,0,Workstation,public,[5001, 5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
call(5000002,_,5001,1001).
call(5000005,_,5002,1002).
call(51,_,5001,48).
call(52,_,5002,46).
```

Figure 6.5: Underlying logic representations of the LAN simulation after refactorings

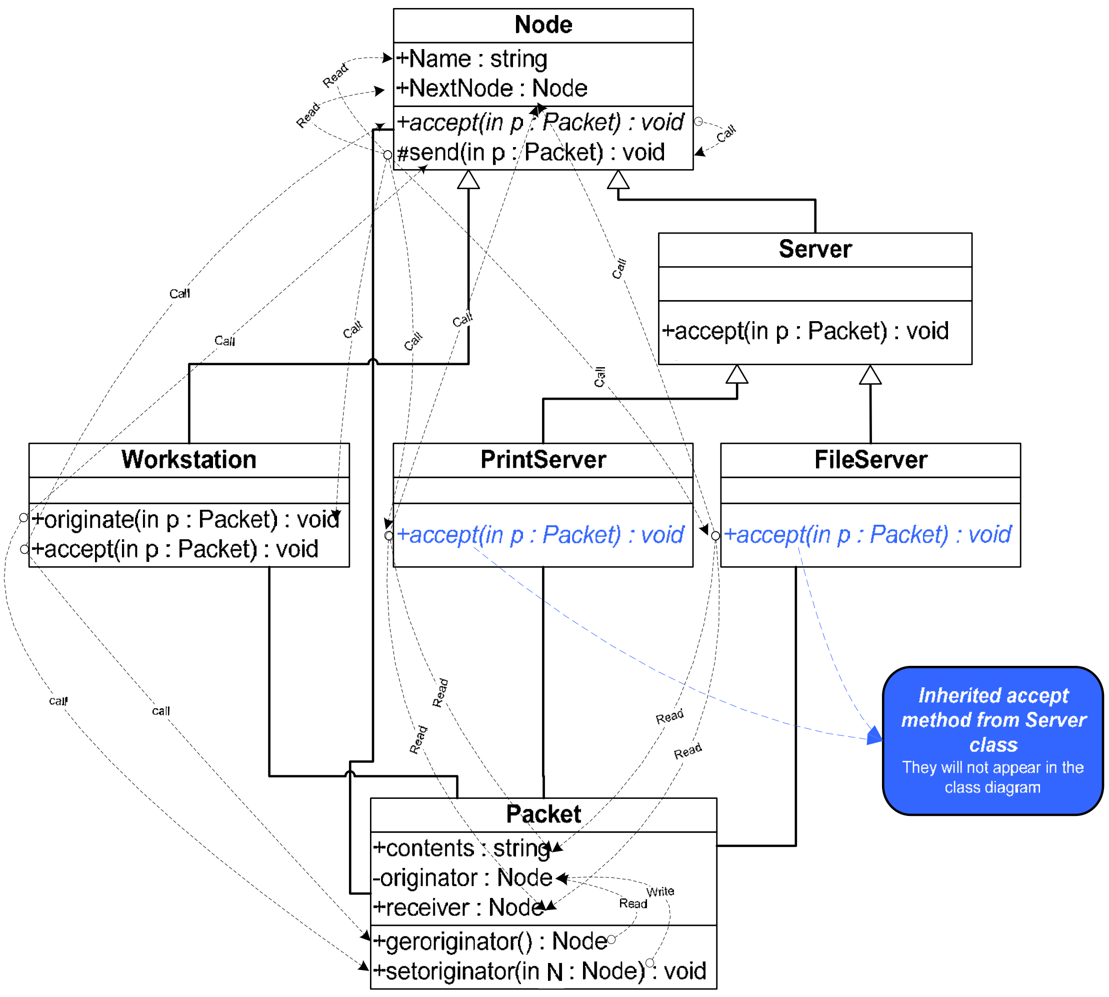

Node
+Name : string
+NextNode : Node
+accept(in p : Packet) : void
#send(in p : Packet) : void
Server
+accept(in p : Packet) : void
Workstation
+originate(in p : Packet) : void
+accept(in p : Packet) : void
PrintServer
+accept(in p : Packet) : void
FileServer
+accept(in p : Packet) : void
Inherited accept method from Server class
They will not appear in the class diagram
Packet
+contents : string
-originator : Node
+receiver : Node
+geroriginator() : Node
+setoriginator(in N : Node) : void
Read
Call
Write

Figure 6.6: A UML class diagram of the LAN simulation after refactoring

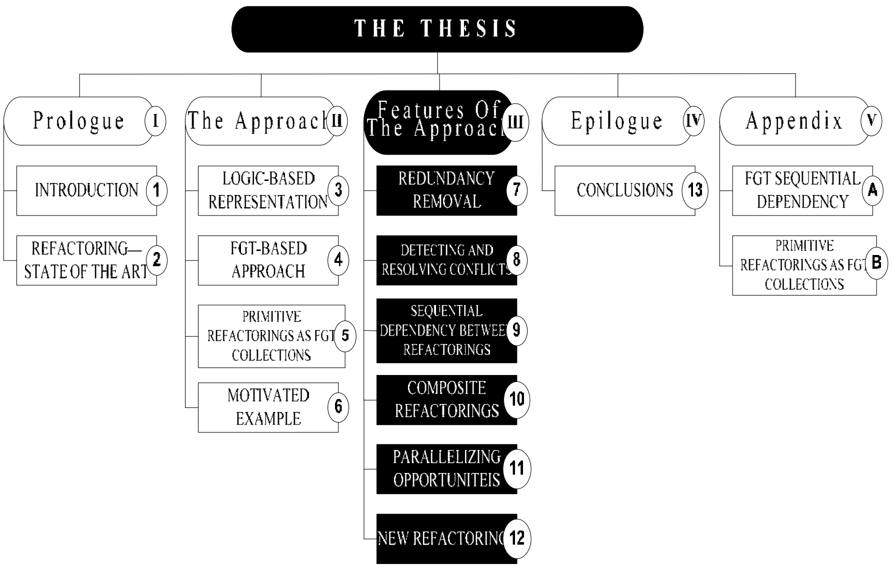
```
package Lan;
class Node {
 public String Name;
 public Node NextNode;
 public void accept(Packet p) {
 this.send(p); }
 protected void send(Packet p) {
 System.out.println(name + nextNode.name);
 this.nextNode.accept(p); }}
class Packet {
 public String contents;
 private Node originator;
 public Node addressee;
 public void setOriginator(Node originator){
 this.originator = originator;}
 public Node getOriginator() {
 return originator;}}
```

```
class Server extends Node{
 public void accept(Packet p) {
 if(p.addressee == this)
   System.out.println(p.contents);
 else super.accept(p); }}
class PrintServer extends Server {}
class FileServer extends Server {}
class Workstation extends Node {
 public void originate(Packet p) {
 p.setOriginator(this);
 System.out.println(this.name+
this.nextNode);
 this.send(p); }
 public void accept(Packet p) {
 if(p.getOriginator() == this)
 System.err.println("no destination");
 else super.accept(p); }}
```

Figure 6.7: A code-level implementation of the LAN simulation after refactoring

Part III

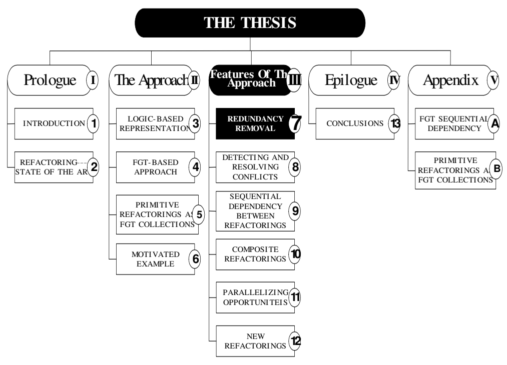
Features Of The Approach


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING—STATE OF THE ART 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORING 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

CHAPTER 7

REDUNDANCY REMOVAL

7.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITIES 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

Applying refactorings on a system can be a time-consuming process, especially when the refactorings are to be applied to a large system. The cost is caused by checking the precondition of the refactoring and by running the required transformation operations on the system. For example, one of the precondition conjuncts of the **deleteMethod** primitive refactoring is that the method should not be referenced anywhere in the system. In this case, the refactoring tool has to check the entire system to look for references to that method. In addition to checking refactoring precondition, some refactorings cause a lot of changes (restructuring) to the system, and this in turn implies executing multiple transformation operations. For example, in the primitive refactoring **moveMethod** described in section 5.3.2.2 many transformation operations are needed to implement the refactoring. The cost is correspondingly higher in the case of composite refactorings, since these may have a significant number of precondition conjuncts that need to be checked, and may significantly restructure the system.

In some cases, a collection of refactorings may embody redundancies. Redundancy occurs whenever a subset of transformation actions undertaken to refactor a system turns out to be unnecessary. In an extreme case, the entire refactoring may have no effect at all on the original system. Redundancy might mean that needless work and effort are done by the refactoring tool, as the following two sections describe.

Previous approaches do not allow for the removal of such redundancies, because refactoring is implemented as a sequence of code blocks (*black box*). No meta-information is available to the refactoring tool to indicate what each part of the code does, and consequently, the tool has no ability to optimise the code.

One of the advantages of dealing with refactoring as a collection of FGTs is that opportunities become available to remove such redundancies. We call this process a *reduction process*. The final effects of the refactoring on the system after the reduction process is the same as the final effects without any reduction. Two types of reductions can be identified: absorbing reductions and cancelling reductions. In the following two sections, each one of the two types will be discussed.

7.2 Absorbing Reduction

This kind of reduction occurs when two FGTs can be absorbed by one that has the same effect as the two. For example, suppose that the user wants to add a new method *m1* in class *P.A*. To do this the following FGT is used:

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

To apply this FGT on the system the refactoring tool has to check its set of precondition conjuncts as described in section 4.2.1.1.B.

Suppose that the user then decides to rename the method *m1* in the class *P.A* to another name *m2*. To do this the following FGT is used:

renameObject(*P, A, m1, _, [], method, m2*)

To apply this FGT on the system the refactoring tool also has to check its set of precondition conjuncts as described in section 4.2.1.2.B.

In accomplishing these tasks, suppose that the refactoring tool carries out the following steps:

a. Check the set of precondition conjuncts of the FGT addObject. Suppose that the refactoring tool performs this check with effort *E1*.

b. Apply the FGT addObject to the system. Suppose that the refactoring tool performs the required transformations with effort *E2*.

c. Check the set of precondition conjuncts of the FGT renameObject. Suppose that the refactoring tool performs this checking with effort *E3*.

d. Apply the FGT renameObject on the system. Suppose that the refactoring tool performs the required transformations with effort *E4*.

The total effort ($T_{effort1}$) required by the refactoring tool to accomplish the previous scenario is therefore:

$$T_{effort1} = E1 + E2 + E3 + E4$$

Alternatively, the refactoring tool can test for redundancies implied by the two FGTs and, if found, make a suitable reduction. To process the scenario, the refactoring tool carries out the following steps:

a. Build the FGT-DAGs of an FGT-list. Suppose that the refactoring tool builds the FGT-DAGs with effort *E5*,

b. Execute the reduction algorithm on the generated FGT-DAGs. In our example, the tool will discover that a redundancy is implicit in the two FGTs. As a result the two FGTs will be absorbed into one FGT that has the same effect as the two:

   addObject(*P, A, m2, _, _, type(basic, void, 0), public, [], method*)

   Suppose that the refactoring tool performs the reduction with effort *E6*.

c. Check the set of the precondition conjuncts of the FGT addObject (with effort *E7*).

d. Apply the FGT on the model (with effort *E8*).

The total effort ($T_{effort2}$) required of the refactoring tool to accomplish the previous scenario is then:

$$T_{effort2} = E5 + E6 + E7 + E8$$

Assume that *E5* and *E6* are likely to be small because they are simple internal processes inside the refactoring tool that, in most of the cases, do not need to reference the underlying representation of the system. Let α =E5 + E6, then

$$T_{effort2} = \alpha + E7 + E8$$

To compare $T_{effort1}$ with $T_{effort2}$ note that $E1 = E7$ and $E2 = E8$ so that

$$T_{effort1} - T_{effort2} = E3 + E4 - \alpha$$

Assuming that α is much smaller than ($E3$ + $E4$), the total effort of the tool with reduction would be much less than the total effort without reduction ($T_{effort2} << T_{effort1}$). However, for a further discussion of these matters refer to 7.7.

Table 7.1 gives the absorbing reductions that may exist between the different pairs of FGTs. Each pair of FGTs that can be reduced is called a *reduction-pair*. The left column of the table shows the reduction-pairs, and the right column of the table shows the suitable FGT that absorbs the pair in the left column. Information in the table is stored as facts in the Prolog database. Examples of these facts are shown in Figure 7.1.

Table 7.1: Absorbing reduction

No	Reduction-Pairs	Absorbed By
1.	renameObject(P,C,M,PR,LT,parameter,X) → renameObject(P,C,M,X,LT,parameter,Y)	renameObject(P,C,M,PR,LT,parameter,Y)
2.	renameObject(P,C,M,_,_,attribute,X) → renameObject(P, C,X,_,_,attribute,Y)	renameObject(P,C,M,_,_,attribute,Y)
3.	renameObject(P,C,M,_,LT,method,X) → renameObject(P,C,X,_,LT,method,Y)	renameObject(P,C,M,_,LT,method,Y)
4.	renameObject(P,C,_,_,_,class,X) → renameObject(P,X,_,_,_,class,Y)	renameObject(P,C,_,_,_,class,Y)
5.	changeOAMode(P,C,M,PR,LT,ObjT,X,Y) → changeOAMode(P,C,M,PR,LT,ObjT,Y,Z)	changeOAMode(P,C,M,PR,LT,ObjT,X,Z)
6.	changeODefType(P,C,M,PR,LT,ObjT,X,Y) → changeODefType(P,C,M,PR,LT,ObjT,Y,Z)	changeODefType(P,C,M,PR,LT,ObjT,X,Z)
7.	addObject(P,C,M,X,T1,T2,T4,T5,parameter) → renameObject(P,C,M,X,T5,parameter,Y)	addObject(P,C,M,Y,T1,T2,T4,T5, parameter)
8.	addObject(P,C,X,_,T2,T3,T5,_,attribute) → renameObject(P,C,X,_,_,attribute,Y)	addObject(P,C,Y,_,T2,T3,T5,_,attribute)
9.	addObject(P,C,X,_,T2,T3,T5,T6,method) → renameObject(P,C,X,_,T6,method,Y)	addObject(P,C,Y,_,T2,T3,T5,T6,method)
10.	addObject(P,X,_,_,T2,T3,T5,_,class) → renameObject(P,X,_,_,_,class,Y)	addObject(P,Y,_,_,T2,T3,T5,_,class)
11.	addObject(P,C,M,PR,T1,X ,T2,T3,ObjT) → changeODefType(P,C,M,PR,T3,ObjT,X,Y)	addObject(P,C,M,PR,T1,Y ,T2,T3,ObjT)
12.	addObject(P,C,M,PR,T1,T2 ,X,T4,ObjT)→ changeOAMode(P,C,M,PR,T4,ObjT,X,Y)	addObject(P,C,M,PR,T1,T2 ,Y,T4, ObjT)
13.	changeOAMode(P,C,M,PR,LT,ObjT,X,Y) → deleteObject(P,C,M,PR,ObjT)	deleteObject(P,C,M,PR,ObjT)
14.	changeODefType(P,C,M,PR,LT,ObjT,X,Y) → deleteObject(P,C,M,PR,ObjT)	deleteObject(P,C,M,PR,ObjT)
15.	renameObject(P,C,M,PR,LT,parameter,PR1) → deleteObject(P,C,M,PR1,LT,parameter)	deleteObject(P,C,M,PR,LT,parameter)
16.	renameObject(P,C,M,_,_,attribute,M1) → deleteObject(P,C,M1,_,_,attribute)	deleteObject(P,C,M,_,LT,attribute)
17.	renameObject(P,C,M,_,LT,method,M1) → deleteObject(P,C,M1,_,LT,method)	deleteObject(P,C,M,_,LT,method)
18.	renameObject(P,C,_,_,_,class,M1) → deleteObject(P,M1,_,_,_,class)	deleteObject(P,C,_,_,_,class)

19.	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*) → renameRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L3*)	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1, M1, PR1, LT1,Totype,Ltype,L3*)
20.	addRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1, Totype,Ltype*) → renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*)	addRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1, PR1,LT1,Totype,Ltype*)
21.	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*) → deleteRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype*)	deleteRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1, PR1,LT1,Totype,Ltype*)

Here is a detailed explanation of each pair of the absorbing reduction:

- Reduction-pairs 1-4 in the table concern the FGT renameObject. If a first FGT renames the *object* element from *name1* to *name2*, and then a second FGT renames the same *object* element from *name2* to *name3,* then these two FGTs can be absorbed by one which will rename the *object* from *name1* to *name3*.

- Reduction-pair 5 concerns the FGT changeOAMode. If a first FGT changes the access mode of the object element from *X* to *Y*, and then a second FGT changes the access mode of the object element form *Y* to *Z* then these two FGTs can be absorbed by one which will change the access mode of the object from *X* to *Z*.

- Reduction-pair 6 is the same as the previous one but for changeODefType.

- Reduction-pairs 7-10 concern the FGTs addObject and renameObject. If the FGT addObject adds a specific object with name *X*, and then a second FGT renameObject changes the name of the same object from *X* to *Y,* then these two FGTs can be absorbed by addObject that adds the object with name *Y* from the beginning.

- Reduction-pair 11 concerns the FGTs addObject and changeODefType. If the FGT addObject adds a specific object with a definition type *X*, and then a second FGT changeODefType changes the definition type of the same object from *X* to *Y* then these two FGTs can be absorbed by one addObject that adds the same object with the definition type *Y* from the beginning.

- Reduction-pair 12 is the same as the previous one but for changeOAMode.

- Reduction-pair 13 concerns the FGTs changeOAMode and deleteObject. If the FGT changeOAMode changes the access mode of specific object from *X* to *Y*, and then a second FGT deleteObject deletes the same object from the system, then there is no need to change the access mode of the object that is going to be deleted in a later stage so these two FGTs can be absorbed by one deleteObject. Note that because the access mode of the object not appears in the FGT deletObject, then in this case, we just remove the FGT changeOAMode from the collection.

- Reduction-pair 14 is the same as the previous one but for changeODefType.
- Reduction-pair 15-18 concern with the FGTs renameObject and deleteObject. If the FGT renameObject changes the name of specific object from *X* to *Y*, and then a second FGT deleteObject deletes object *Y* from the system then in this case, there is no need to change the name of the object that is going to be deleted in a later stage so these two FGTs can be absorbed by one deleteObject that will delete the object with name *X* (old name).
- Reduction-pair 19 is the same as reduction-pairs 1-4 but for renameRelation.
- Reduction-pair 20 is the same as reduction-pairs 7-10 but for FGTs addRelation and renameRelation.
- Reduction-pair 21 is the same as reduction-pairs 15-18 but for FGTs renameRelation and deleteRelation.

7.3 Cancelling Reduction

This kind of reduction occurs when two FGTs cancel each other. For example, suppose that a user adds a new method *m1* in class *P.A*. To do this, he uses the following FGT:

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

To apply the FGT, the refactoring tool has to check the relevant set of precondition conjuncts. Suppose that the method *m1* in the class *P.A* is subsequently deleted. To do this the following FGT is used:

deleteObject(*P, A, m1, _, [], method*)

It is clear that there is a reduction between the two FGTs. The refactoring tool will discover that there is a cancelling reduction between the two FGTs. As a result, the two FGTs will be removed from the refactoring collection. In this case, the only effort needed of the tool is to build the FGT-DAGs and carry out the reduction process. If we assume that *E1*, *E2* stands for checking precondition conjuncts and applying FGT addObject and *E3*, *E4* stands for checking precondition conjuncts and applying FGT deleteObject then the effort of the refactoring tool without reduction will be:

$$T_{effort1} = E1 + E2 + E3 + E4$$

While the effort of the refactoring tool in case of using the reduction will be

$$T_{effort2} = \alpha$$

where α is the effort to build the FGT-DAGs and execute reduction process. As assumed before that α is small because it is an internal process within the tool. Therefore, it can be concluded that

$$T_{effort2} << T_{effort1}.$$

Table 7.2 gives the various possibilities for the cancelling reductions between different pairs of FGTs. Information in the table is stored as facts in the Prolog database, part of these facts are shown in Figure 7.1.

Table 7.2: Cancelling reduction

No	Reduction-Pairs
1.	deleteObject(*P,C,M,PR,LT,ObjT*) → addObject(*P,C,M,PR,_,_,_,LT,ObjT*)
2.	addObject(*P,C,M,PR,_,_,_,LT,ObjT*) → deleteObject(*P,C,M,PR,LT,ObjT*)
3.	deleteRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*) → addRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*)
4.	addRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*) → deleteRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*)

Here is a detailed explanation of each pair of the cancelling reduction:

- Reduction-pairs 1-2 in the table concern the FGTs deleteObject and addObject. To delete an *object* element from the system using FGT deleteObject and then to add the same *object* to the system using FGT addObject clearly has no effect on the system. The same also apply when add an *object* element to the system using FGT addObject and then to delete the same *object* element from the system using FGT deleteObject. As a result, the two FGTs need to be removed by the refactoring tool.

- Reduction-pairs 3-4 in the table are similar to the previous reduction-pair, but involve addRelation and deteteRelation.

```
…………………….
reduction(renameObject(P,C,_,_,_,class,X),renameObject(P,X,_,_,_,class,Y),
renameObject(P,C,_,_,_,class,Y)).

reduction(addObject(P,C,X,_,T2,T3,T4,T5,T6,method), renameObject(P,C,X,_,
T6,method,Y), addObject(P,C,Y,_,T2,T3,T4,T5,T6,method)).

reduction(addObject(P,C,M,PR,_,_,_,_,LT,ObjT),deleteObject(P,C,M,PR,LT,ObjT),
'Cancel Both').
……………………….
```

Figure 7.1: Part of the reduction facts as implemented in Prolog

7.4 Advantages of Reduction Process

The reduction process has the following advantages:

1. The number of FGTs and number of refactoring precondition conjuncts are reduced, thus increasing the efficiency of refactoring. Clearly, when an FGT is cancelled or absorbed by the reduction process, then its set of precondition conjuncts will also be cancelled or absorbed. One advantage of distributing the precondition conjuncts of the refactoring into two levels (FGT-level and refactoring-level) is the ability to cancel or absorb these precondition conjuncts by the reduction process.

2. The number of sequential dependencies between the different FGTs inside the refactoring will be reduced. This will increase the number of FGT-DAGs for that refactoring which means that the parallelizing opportunities at the time of refactoring will be increased.

3. Pseudo-conflicts may be eliminated. For example, suppose that we have the following two refactorings:

R_x: {…………..
…………..
addObject (*P, A , m1, _, _, type(basic, void, 0), public, [], method*)
…………..
deleteObject (*P, A, m1, [], method*)
…………..
…………..}

R_y: {…………..
…………..
addObject (*P, A, m1, _, _, type(basic, void, 0), public, [], method*)
…………..
..…………...}

Trying to apply refactorings R_x and R_y concurrently on the system (or in the order R_y then R_x) could cause a conflict between the two refactorings because both of the two refactorings try to add the same method *m1* in the class *P.A* (conflicts between concurrent refactorings will be discussed in the next chapter). To solve this conflict one of the two refactorings could be cancelled or R_x could be executed before R_y.

In fact, the conflict between refactorings R_x and R_y is pseudo-conflict. If a reduction process discovers that there is a cancelling reduction between FGTs addObject and deleteObject in R_x and removes these two FGTs from the collection of FGTs of refactoring R_x, then there will be no conflict between R_x and R_y.

7.5 Reduction Algorithm

A reduction algorithm has been developed that takes an arbitrary FGT-DAG as input, and removes all redundancies in this data structure. This algorithm may be invoked to remove possible redundancies from some FGT-DAGs that represent a refactoring. Its use also will be seen in chapter 10 in the context of composite refactorings.

It can easily be verified that FGTs in each reduction-pair are sequentially dependent. If they appear as adjacent nodes in an FGT-DAG then they may be redundant. The reduction algorithm is based on this. It simply traverses an FGT-DAG and searches for adjacent reduction-pairs. When one is found, then the algorithm makes the suitable reduction and appropriately restructures the rest of nodes in that FGT-DAG. As will be seen, this reduction may result in new FGT-DAGs being created out of parts of the original FGT-DAG in which the redundancy was found.

As mentioned before, the refactoring may consist of more than one FGT-DAG. Since these are sequentially independent, the different instances of the reduction algorithm can work concurrently on each FGT-DAG.

Algorithm 7.1 gives the pseudo-code for the **reduction** algorithm. The algorithm takes as a parameter an FGT-DAG. It traverses the FGT-DAG from root to leaves in a depth-first fashion, searching for occurrences of reduction-pairs ($node_i$ and $node_j$) using the reduction facts. If a reduction-pair is found then the corresponding reduction is made and the links between the remaining FGTs in the FGT-DAG are changed properly.

If one of the cancelling reduction-pairs is found, then the reduction-pair (*node*$_i$, *node*$_j$) will be removed from the FGT-DAG. All links into and out of *node*$_i$ and *node*$_j$ will also be removed. In addition, the algorithm will check for sequential dependencies between each father of *node*$_i$ and the sons of both *node*$_i$ and *node*$_j$. If there is a sequential dependency then a new link will be created between the relevant father and son nodes. The same is done with respect to each father of *node*$_j$ and sons of both *node*$_i$ and *node*$_j$. As will be illustrated below, the above process may result in one or more FGT-DAGs being formed from parts of the old FGT-DAG.

If an absorbing reduction-pair is found, then the reduction-pair (*node*$_i$, *node*$_j$) will be removed from the FGT-DAG as well as all links related to these nodes. A new node called *node*$_x$ as specified by the relevant reduction fact is then inserted into the FGT-DAG. Again, the algorithm will check for sequential dependencies, in this case between each father of *node*$_i$ (and of *node*$_j$) and the newly created *node*$_x$. If a sequential dependency is found, then a new link will be created between the relevant father node and *node*$_x$. Similarly, a check for sequential dependencies will be made between *node*$_x$ and each son of *node*$_i$ and each son of *node*$_j$. Again, wherever a sequential dependency is found, a new link will be created between *node*$_x$ and the relevant son node. Also in this case, it is possible that the above process results in one or more FGT-DAGs being formed from parts of the old FGT-DAG.

Algorithm 7.1 (Reduction algorithm)

```
reduction ( IN-DAG )
Input:    IN-DAG: An FGT-DAG
Output:   RED-DAGS: A redundancy-free set of FGT-DAGs

Insert IN-DAG into RED_DAGS
For each unexamined pair of adjacent nodes (node_i, node_j) in RED-DAGS do {
    //Search reduction facts for a match between (node_i, node_j)
    If (node_i, node_j) is a cancelling reduction-pair
      then {
              Let F = set of father nodes of node_i and father nodes of node_j (excluding node_i)
              Let S = set of son nodes of node_i (excluding node_j) and son nodes of node_j

              Let FS = F X S

              For each pair (node_f , node_s ) in FS do {
                If (node_f , node_s ) sequentially dependent
                  then insert link from node_f to node_s
                } //end For
                Remove (node$_i$, node$_j$ ) from IN-DAG
                Remove all links into and out of node$_i$, and node$_j$
    } //end If
    Else If (node$_i$, node$_j$) is an absorbing pair
        then {
                Let node$_x$=absorbing FGT of (node$_i$, node$_j$ )
                Let F = set of father nodes of node$_i$ and father nodes of node$_j$ (excluding node$_i$ )
                Let S = set of son nodes of node$_i$ (excluding node$_j$) and son nodes of node$_j$
                For each node$_f$ in F do {
                    If (node$_f$, node$_x$) sequentially dependent
                      then insert link from node$_f$ to node$_x$
                } //end for
                For each node$_s$ in S do {
                    If (node$_x$, node$_s$) sequentially dependent
                      then insert link from node$_x$ to node$_s$
                } //end For
                Remove (node$_i$, node$_j$) from IN-DAG
    } //end Else If
    Collect all FGT-DAGs produced by the foregoing into RED-DAGS
} //end For
Return RED-DAGS
```

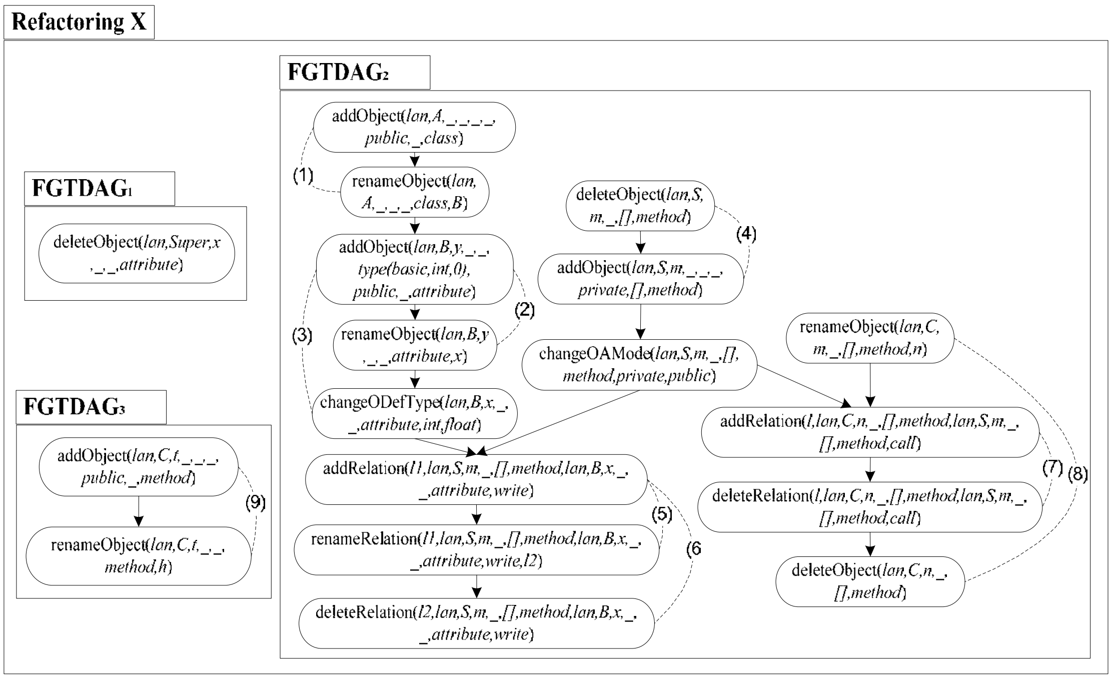
Note that the for-loop in the algorithm is not specific about the order in which adjacent nodes are examined. It does require, however, that new adjacent pairs that may be added into the FGT-DAGs in RED-DAGS have to be examined. The actual order to be used is an implementation issue. In the prototype tool, a top-down approach has been followed.

7.6 Example

To illustrate the reduction idea, a fictitious example as shown in Figure 7.2 is used. In the example, refactoring X consists of three independent FGT-DAGs (FGT-DAG$_1$, FGT-DAG$_2$ and FGT-DAG$_3$). The **reduction** algorithm will work on each one of the three FGT-DAGs separately.

For FGT-DAG$_2$ for example, the order in which the algorithm examines node pairs is indicated by the numbering on the dashed lines of Figure 7.2(a). Figure 7.2(b) shows the reduction action that the algorithm takes for each reduction-pair.

Figure 7.3 shows refactoring X after being reduced. Note that the number of reductions depends on the type of FGTs in the refactoring. The number of FGTs in the example has been reduced from 18 FGTs to 6 FGTs. Note also that the number of FGT-DAGs has been increased from 3 to 5 after reduction.


Refactoring X
FGTDAG1
deleteObject(lan,Super,x,_,_,attribute)
FGTDAG3
addObject(lan,C,f,_,_,_,public,_,method)
renameObject(lan,C,f,_,_,method,h)
(9)
FGTDAG2
addObject(lan,A,_,_,_,_,public,_,class)
renameObject(lan,A,_,_,_,class,B)
(1)
addObject(lan,B,y,_,_,type(basic,int,0),public,_,attribute)
(2)
renameObject(lan,B,y,_,_,attribute,x)
(3)
changeODefType(lan,B,x,_,_,attribute,int,float)
deleteObject(lan,S,m,_,[],method)
addObject(lan,S,m,_,_,_,private,[],method)
(4)
changeOAMode(lan,S,m,_,[],method,private,public)
renameObject(lan,C,m,_,[],method,n)
addRelation(l1,lan,S,m,_,[],method,lan,B,x,_,_,attribute,write)
renameRelation(l1,lan,S,m,_,[],method,lan,B,x,_,_,attribute,write,l2)
deleteRelation(l2,lan,S,m,_,[],method,lan,B,x,_,_,attribute,write)
(5)
(6
addRelation(l,lan,C,n,_,[],method,lan,S,m,_,[],method,call)
deleteRelation(l,lan,C,n,_,[],method,lan,S,m,_,[],method,call)
deleteObject(lan,C,n,_,[],method)
(7)
(8)

(a)

Reduction Actions

(1) Absorbed by addObject(*lan*,*B*,_,_,_,_,*public*,_,*class*)
(2) Absorbed by addObject(*lan*,*B*,*x*,_,_,*type*(*basic*,*int*,*0*),*public*,_,*attribute*)
(3) Absorbed by addObject(*lan*,*B*,*y*,_,_,*type*(*basic*,*float*,*0*),*public*,_,*attribute*)
(4) Delete both
(5) Absorbed by addRelation(*l2*,*lan*,*S*,*m*,_,[],*method*,*lan*,*B*,*x*,_,_,*attribute*,*write*)
(6) Delete both
(7) Delete both
(8) Absorbed by deleteObject(*lan*,*C*,*n*,_,[],*method*)
(9) Absorbed by addObject(lan,*C*,*h*,_,_,_, *public*,[],*method*)

(b)

Figure 7.2: Reduction inside refactoring

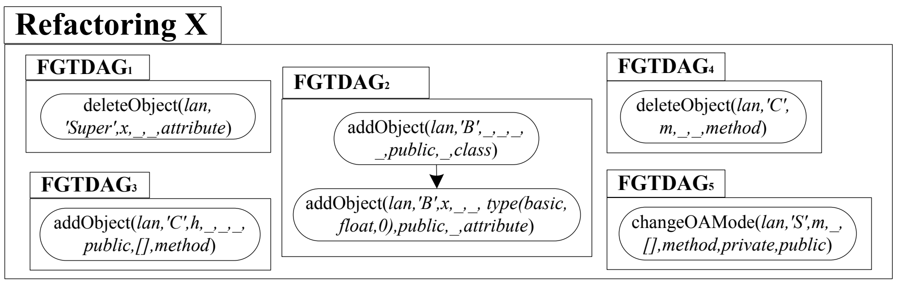

Refactoring X
FGTDAG1
deleteObject(lan, 'Super',x,_,_,attribute)
FGTDAG2
addObject(lan,'B',_,_,_,_,public,_,class)
addObject(lan,'B',x,_,_, type(basic, float,0),public,_,attribute)
FGTDAG3
addObject(lan,'C',h,_,_,_, public,[],method)
FGTDAG4
deleteObject(lan,'C', m,_,_,method)
FGTDAG5
changeOAMode(lan,'S',m,_, [],method,private,public)

Figure 7.3: Refactoring X after reduction

7.7 Efficiency Considerations

The possibility that a given FGT-DAG may embody redundancies, and that these may be removed, is certainly of theoretical interest. How such redundancies may come about in practice is an open question. It may be, for example, that they are specified by a naive user attempting to use FGTs to specify a transformation or refactoring. Alternatively, redundancies may arise in a multi-user environment where different FGT-lists are merged.

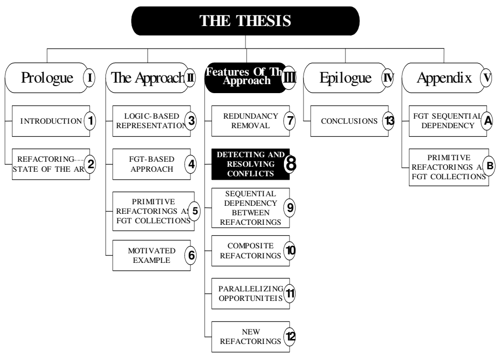
The question of whether or not it is efficient to remove redundancies from an FGT-DAG is also context-dependent. There are definite gains to be had in reducing the number of changes to the underlying system. If the system is large, its underlying representation is correspondingly large and unnecessarily searches into the data are best avoided. In contrast, the database of facts recording redundancy pairs and sequential dependencies is not system-dependent and can be accessed relatively efficiently for the purposes of setting up or changing FGT-DAGs. However, the overall cost of setting up FGT-DAGs and reducing them is also dependent on the originating FGT-list.

Notwithstanding these context-dependent efficiency considerations, in forthcoming chapters, all relevant FGT-DAGs will be considered to be redundancy-free.

Chapter 8

DETECTING AND RESOLVING CONFLICTS

8.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of Th Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
LOGIC-BASED REPRESENTATIO 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS A FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS A FGT COLLECTIONS B

This chapter and the next consider two refactorings, R_i and R_j. How these refactorings came into being is not relevant. What is assumed to be known is the pre- and postconditions of the two refactorings. In addition, it is assumed that the internal composition of each refactoring is known in terms of a set of FGT-DAGs. The two respective chapters then enquire into the question of whether the two refactorings are related in a manner that constrains the way in which they can be applied to a system. There are three possible answers to this question.

1. They are entirely unrelated. In this case, they can be applied in any order to a system (However, without information about precisely what changes are made to the system while they are being applied, it cannot be asserted that they may safely run concurrently.)

2. There is some order in which the two refactorings have to be applied—either R_i then R_j or vice versa. This is the subject of enquiry in chapter 9.

3. It is not possible to apply both refactorings on any system. This is the subject of enquiry in the present chapter—namely, the matter of detecting conflicts, and possibly resolving them.

Conflict between refactoring R_i and refactoring R_j occurs when it is the case that applying them in a given order will make the later one inapplicable. By this, is meant that when the first refactoring is applied to the system it will change the state of the system in a way that makes the precondition of the second one inapplicable. Thus, the postcondition of the first will conflict with the precondition of the second.

For example, suppose in a multi-developer environment, two developers try to apply refactorings R_i and R_j to the same system. Assume that the system has a package P with one class C. Assume—as shown in Figure 8.1—that part of the transformations that the two refactorings intend to make on the system are as the follows: refactoring R_i adds a new class A

in the package P and refactoring R_j changes the name of the existing class C in the package P to a new name A. Note that part of the precondition conjuncts of the two refactorings is the non-existence of a class with name A in the package P.

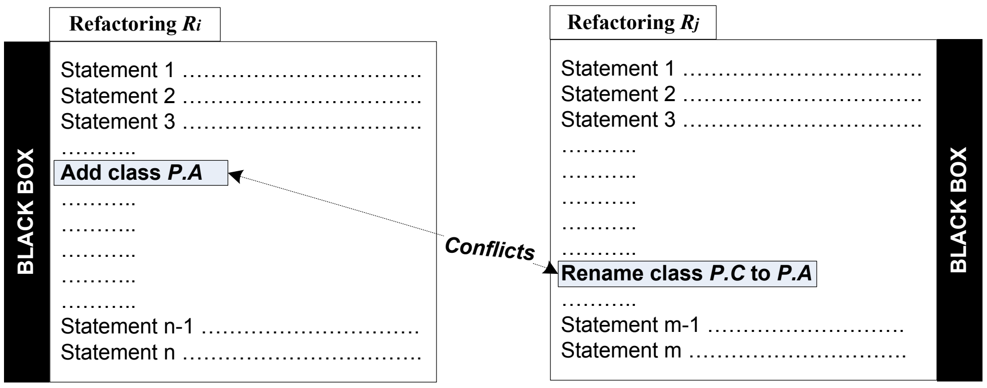


Figure 8.1: Conflicts between refactorings R_i & R_j

In the previous example, three possible scenarios may be envisaged:

1. Apply refactoring R_i then refactoring R_j. In this case, the tool will check the precondition conjuncts of R_i, discovering that they are satisfied because class A is not defined in the package P. Therefore, the tool will apply refactoring R_i to the system which means that the class A will be defined in the package P by this refactoring. Then at the time of applying refactoring R_j, the tool will check the precondition conjuncts of R_j, discovering that they do not hold because class A is defined now in the package P.

2. Apply refactoring R_j then refactoring R_i. In this case, the tool will check the precondition conjuncts of R_j, discovering that they are satisfied because class A is not defined in the package P. Therefore, the tool will apply refactoring R_j to the system which means that the class A will be defined in the package P by this refactoring. Then at the time of applying refactoring R_i, the tool will check the precondition conjuncts of R_i, discovering that they do not hold because class A is defined now in the package P.

3. Apply refactorings R_i and R_j simultaneously. In this case when the refactoring tool checks the precondition conjuncts of refactoring R_i and R_j, it will find that the precondition conjuncts of the two refactorings are satisfied because class A is not defined in package P at that time. Then the tool will start applying the two refactorings to the system simultaneously which will end up with an inconsistency because at the end of the two refactorings, package P will have two classes with the same name A.

To avoid such problems, a refactoring tool should have the capability to detect and resolve conflicts that may occur between multiple refactorings.

Tools based on previous refactoring approaches can potentially be designed to detect that there is a conflict between two refactorings. One approach is based on checking the pre- and postconditions of the two refactorings [38, 39, 52, and 55]. In the example presented in Figure 8.1, part of the precondition conjuncts of refactorings R_i and R_j is the non-existence of class *A* in package *P*. In addition, part of the postcondition conjuncts of the two refactorings is the existence of class *A* in package *P*. From this information, the refactoring tool could infer that there is a conflict between the two refactorings. Another approach proposed in the graph transformation community. The approach is based on the technique of critical pair analysis [55, 57, and 58].

However, it is difficult for such a tool to detect which specific parts of the two refactorings cause the conflict, and therefore to take possible corrective steps that could potentially resolve the conflict.

8.2 Conflicts in FGT-Based Approach

To detect and resolve conflicts between multiple refactorings in an FGT-based approach, it is sufficient to detect and resolve conflicts at the level of those FGTs which make up these refactorings. Consider the refactorings R_i and R_j discussed in the previous example but now shown in Figure 8.2 as a collection of FGTs ordered in FGT-DAGs. In such a scenario, the refactoring tool can check for conflicts between every pair of FGTs in the two refactorings.

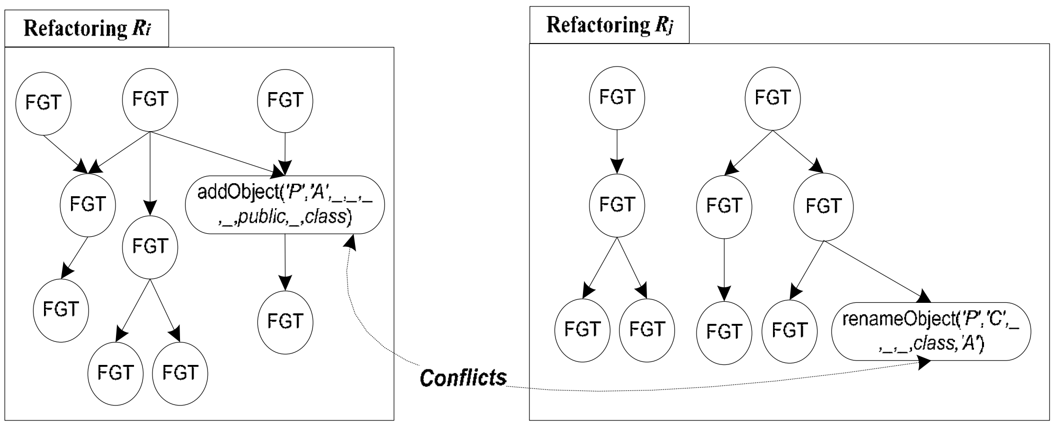

Refactoring Ri
FGT
addObject('P','A',_,_,_,_,public,_,class)
Conflicts
Refactoring Rj
renameObject('P','C',_,_,_,class,'A')

Figure 8.2: Conflict detection in FGT-based approach

This approach is close to what is called the **operation-based merge** approach [16, 21, 32, 50, 62, and 80] that is used to find conflicts between multiple versions of software that need to be merged after being changed/evolved by multiple developers. To detect the merge conflicts in such approaches, there is no need to compare all versions in their entirety—it suffices to compare only the evolution operations that have been applied to obtain each of the versions. In the present context, these evolution operations are comparable to FGTs that make up the refactorings. The literature suggests that this **operation-based merge** approach is more efficient and solves various problems that occur in other approaches (such as the **text-based merge** approach [32, 44, 45, and 48], in which software artifacts are considered as text or binary files). It is, however, out of the scope of this thesis to go into the details of these different merge approaches.

Should a refactoring tool allow a naive user to define a set of FGT-DAGs as constituting a new primitive refactoring (as explained in chapter 12), it is conceivable that these FGTs might contain mutual conflicts. The tool could, in principle, be designed to trace and report such conflicts. However, for the purposes of this thesis, it will be assumed that FGTs (and thus the associated FGT-DAGs) that make up a primitive refactoring do not conflict with one another. Nevertheless, the possibility arises that multiple primitive refactorings might be submitted to the tool for implementation. The tool ought to be able to detect and resolve conflicts that might exist between two (or more) such refactorings.

The various possibilities of conflicts that may occur between different FGTs have been pre-catalogued, as shown in Figure 8.3 and explained in more detail in Tables 8.1 and 8.2. This information is stored as facts in the Prolog database, examples of which are shown in Figure 8.4.

The arcs with arrows at both ends in Figure 8.3 represent bi-directional conflicts—i.e. conflicts that may occur between FGTs in both directions. Both directions mean that applying the two FGTs in either order will cause a conflict. For example, the following two FGTs obviously have a conflict in both directions:

FGT_1: addObject(*P, A, _, _, _, _, public, _, class*)
FGT_2: renameObject(*P, C, _, _, _, class, A*)

Applying FGT_1 first will prohibit applying FGT_2. This is because FGT_1 will add a new class with name *A* to the package *P* and after that, FGT_2 will try to rename another class *C* in package *P* to a new name *A*. Clearly, applying FGT_2 first will also prohibit subsequently applying FGT_1.

The arcs with a single arrow at one end in Figure 8.3 represent uni-directional conflicts—i.e. conflicts that occur between two FGTs in one direction only. Consider, for example:

FGT$_1$: renameObject (*P, C, _, _, _, class, A*)
FGT$_2$: deleteRelation (*isa, P, C, _, _, _, class, P, D, _, _, _, class, extends*)

Clearly, applying FGT$_1$ first will prohibit applying FGT$_2$ because after changing the name of the class *C* to *A* as per FGT$_1$, FGT$_2$ will not be able to find class *C*—i.e. its set of precondition conjuncts are no longer satisfied. On the other hand, applying FGT$_2$ first will not cause any conflict.

To resolve the conflict after being detected, a resolution procedure is defined for each type of conflict. Conflicts between the different FGTs (conflict-pairs) are categorized into three groups according to the approach used to resolve these conflicts:

1. **Ordering-conflicts** refer to conflicts that can be resolved by applying the two refactorings in a specific order.

2. **Cancelling-conflicts** refer to conflicts that can be resolved by cancelling (withdrawing) one of the two refactorings. The developer will be asked to choose one of the two refactorings to be cancelled.

3. **Removable-conflicts** refer to conflicts that can, in principle, be resolved by modifying one of the two FGTs that participate in the conflict. Suppose that FGT$_x$ is from refactoring X and FGT$_y$ is from refactoring Y; and suppose that the developer is asked to modify FGT$_x$. In doing so, the following should be taken into account:

   a. All FGTs that sequentially depend on FGT$_x$ (i.e. descendants of FGT$_x$ in the FGT-DAG of refactoring X) should also be modified to reflect the changes done on FGT$_x$. However, these changes should not introduce new sequential dependencies or redundancies in the FGT-DAGs of refactoring X.

   b. Changes that the developer makes on FGT$_x$ should not produce new conflicts with ancestors of FGT$_x$ in the relevant FGT-DAG.

   c. Changes that the developer makes on FGT$_x$ should not produce conflicts with FGTs located in different FGT-DAGs of refactoring X—i.e. the FGTs constituting refactoring X should remain conflict-free.

d. The modified FGT$_x$ should not have a conflict with any FGTs in refactoring Y that have already been checked to date.

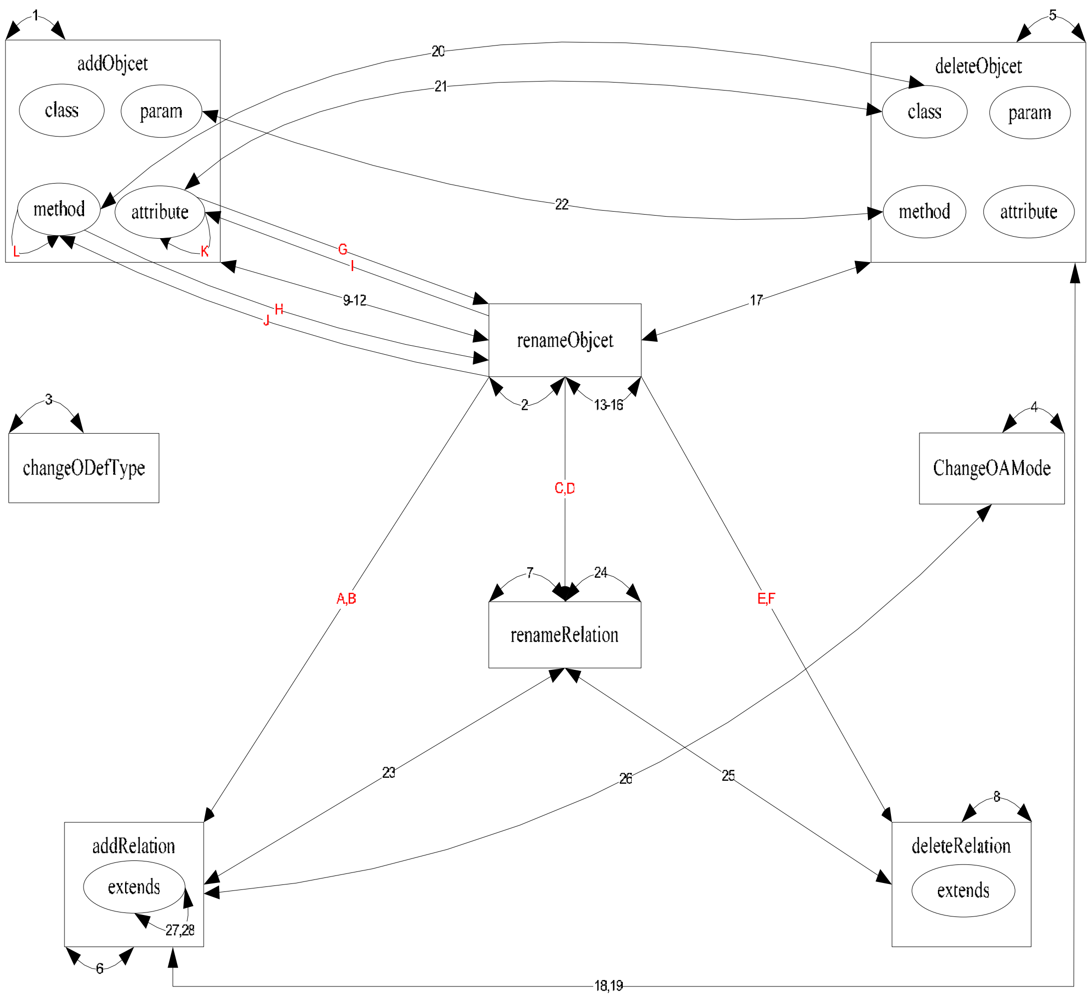


Figure 8.3: Possible conflicts between FGTs

```
..........................
fgtConflict(addobject(P,C,M,X,_,_,_,_,_,OT), addObject(P,C,M,X,_,_,_,_,_,OT)).
fgtConflict(renameObject(P,C,_,_,_,attribute,X), renameObject(P,C,_,_,_,attribute,X)).
fgtConflict(addObject(P,C,X,_,_,_,_,_,_,attribute), deleteObject(P,C,_,_,_,class)).
fgtConflict(changeOAMode(P,C,M,PR,LT,OT,X,Y), addRelation(_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_)).
fgtConflict(renameObject(P,C,M,PR,LT,OT,X), deleteRelation(_,P,C,M,PR,LT,OT,_,_,_,_,_,_,_)).
..........................
```

Figure 8.4: A Selection of fgtConflict facts as implemented in Prolog

It is feasible to identify FGT pairs that constitute removable-conflicts. It is also possible to offer guidelines about how one of the FGTs in the pair may be changed without taking account of the overall context of the FGTs in the pair. However, there is no guarantee that a removable-conflict can indeed be resolved in *every specific context*. It is beyond the scope of this thesis to explore conditions in the surrounding context of a removable-conflict pair that will guarantee that the conflict can indeed be removed. Also left as a matter for future research, is the associated problem of algorithmically resolving removable-conflicts.

8.3 FGT's Conflicts-Pairs

In the following two subsections (8.4.1 and 8.4.2), a detailed description of each type of the bi-directional and uni-directional conflict is given. To clarify the discussion, a simplified UML class diagram of a *College* system is used. The system, shown in Figure 8.5, is in a package called *College* and has three classes: *Teacher*, *Student* and *PostGradStudent*.

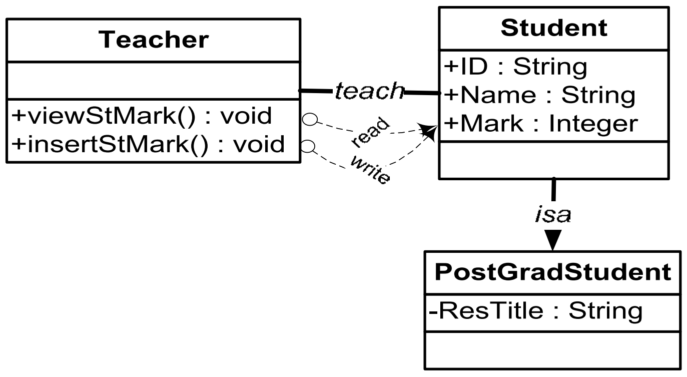


Figure 8.5: A simplified UML class diagram of a college system

8.3.1 Bi-Directional Conflict

A bi-directional conflict is a conflict that may occur between FGTs in both directions. Both directions mean that applying the two FGTs in either order will cause a conflict. In the following, a discussion of each bi-directional conflict catalogued in Table 8.1 is given. It will be seen that these conflicts are never classifiable as ordering-conflicts—only as cancelling or removable-conflicts.

Table 8.1: Bi-directional FGT Conflict-Pairs

No	FGT$_x$	FGT$_y$
1.	addobject(P,C,M,X,_,_,_,_,OT)	addObject(P,C,M,X,_,_,_,_,OT)
2.	renameObject(P,C,M,PR,LT,OT,_)	renameObject(P,C,M,PR,LT,OT,_)
3.	changeODefType(P,C,M,PR,LT,ObjT,X,_)	changeODefType(P,C,M,PR,LT,ObjT,X,_)
4.	changeOAMode(P,C,M,PR,LT,ObjT,X,_)	changeOAMode(P,C,M,PR,LT,ObjT,X,_)
5.	deleteObject(P,C,M,PR,LT,ObjT)	deleteObject(P,C,M,PR,LT,ObjT)
6.	addRelation(RelL,P,C,M,PR,FLT,Ftype,P1,C1,M1 ,PR1,TLT,Totype,Ltype)	addRelation(RelL,P,C,M,PR,FLT,Ftype,P1,C1, M1,PR1,TLT,Totype,Ltype)
7.	renameRelation(X,P,C,M,PR,FLT,Ftype,P1, C1,M1,PR1,TLT,Totype,Ltype,_)	renameRelation(X,P,C,M,PR,FLT,Ftype,P1,C1, M1,PR1,TLT,Totype,Ltype,_)
8.	deleteRelation(RelL,P,C,M,PR,FLT,Ftype, P1,C1,M1,PR1,TLT,Totype,Ltype)	deleteRelation(RelL,P,C,M,PR,FLT,Ftype,P1,C1,M1,P R1,TLT,Totype,Ltype)
9.	addObject(P,C,M,X,_,_,_,LT,parameter)	renameObject(P,C,M,_, LT,parameter,X)
10.	addObject(P,C,X,_,_,_,_,_,attribute)	renameObject(P,C,_,_,_,attribute,X)
11.	addObject(P,C,X,_,_,_,_,LT,method)	renameObject(P,C,_,_, LT,method,X)
12.	addObject(P,X,_,_,_,_,_,_,class)	renameObject(P,_,_,_,_,class,X)
13.	renameObject(P,C,M,_, LT,parameter,X)	renameObject(P,C,M,_, LT,parameter,X)
14.	renameObject(P,C,_,_,_,attribute,X)	renameObject(P,C,_,_,_,attribute,X)
15.	renameObject(P,C,_,_, LT,method,X)	renameObject(P,C,_,_, LT,method,X)
16.	renameObject(P,_,_,_,_,class,X)	renameObject(P,_,_,_,_,class,X)
17.	renameObject(P,C,M,PR,LT,OT,X)	deleteObject(P,C,M,PR,LT,OT)
18.	addRelation(_,P,C,M,PR,LT,Ftype,_,_,_,_,_,_,_)	deleteObject(P,C,M,PR,LT,Ftype)
19.	addRelation(_,_,_,_,_,_,_,_,P,C,M,PR,TL,Ttype,_)	deleteObject(P,C,M,PR,TL,Ttype)
20.	addObject(P,C,X,_,_,_,_,LT,method)	deleteObject(P,C,_,_,_,class)
21.	addObject(P,C,X,_,_,_,_,_,attribute)	deleteObject(P,C,_,_,_,class)
22.	addObject(P,C,M,X,_,_,_,LT,parameter)	deleteObject(P,C,M,_,LT,method)
23.	renameRelation(X,P,C,_,_,_,class,P1,C1,_,_,_, class,association,Y)	addRelation(Y,P,C,_,_,_,class,P1,C1,_,_,_,class, association)
24.	renameRelation(_,P,C,_,_,_,class,P1,C1,_,_,_, class,association,X)	renameRelation(_,P,C,_,_,_,class,P1,C1,_,_,_,class, association,X)
25.	renameRelation(X,P,C,M,PR,FLT,Ftype,P1, C1,M1,PR1,TLT,Totype,LType,Y)	deleteRelation(X,P,C,M,PR,FLT,Ftype,P1,C1,M1,PR1, TLT,Totype,LType)
26.	changeOAMode(P,C,M,PR,LT,OT,X,Y)	addRelation(_,_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_)
27.	addRelation(_,P,C,_,_,_,class,P1,C1,_,_,_,class, extends)	addRelation(_,P2,C2,_,_,_,class,P1,C1,_,_,_,class, extends)
28.	addRelation(_,P,C,_,_,_,class,P1,C1,_,_,_,class, extends)	addRelation(_,P1,C1,_,_,_,class,P,C,_,_,_,class, extends)

- Conflicts 1-8 between FGT$_x$ and FGT$_y$ in the table occur in the case that FGT$_x$ and FGT$_y$ are the same, applying the first one on the system will prohibit applying the second one. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT$_x$: addobject(*College, Student, Age,* _, _, _, _, _, *attribute*)

  FGT$_y$: addobject(*College, Student, Age,* _, _, _, _, _, *attribute*)

It is clear that the two FGTs try to add the same attribute *Age* to the class *Student*—something that cannot happen more than once.

In general, it can easily be seen that attempting to apply any FGT more than once in succession will always cause a conflict. Indeed, it is in the very nature of an FGT to *transform* the state of a system which satisfies its precondition to a *different* state in which the precondition is no longer satisfied.

These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflicts 9-12 between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to add an object X to the system and FGT$_y$ tries to change the name of another object to X, or *vice versa*. This means that the system will have two objects with the same name X which is prohibited.

  The type of these conflicts is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs. For example, in the *College* system trying to apply the following two FGTs successively in either order will cause a conflict:

  FGT$_x$: addobject(*College, Teacher, listSTMarks, _, _, _, _, [], method*)
  FGT$_y$: renameObject(*College, Teacher, viewSTMark, _, [], method, listSTMarks*)

  The conflict could perhaps be resolved by choosing to rename the *viewSTMark* method to, say, *displaySTMark*, should the context permit this.

- Conflicts 13-16 between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to change the name of an object *X* to a new name *Y* and FGT$_y$ tries to change the name of another object *Z,* defined in the same scope as object *X,* to the same new name *Y*. This means that the system will have two objects with the same name *Y* defined within the same scope which is prohibited. This would happen if and attempt was made to apply the following FGTs:

  FGT$_x$: renameObject(*College, Stuednt, ID, _, _, attribute, StPinf*)
  FGT$_y$: renameObject(*College, Stuednt, Name, _, _, attribute, StPinf*)

  The type of these conflicts is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 17 between FGT$_x$ and FGT$_y$ occurs when FGT$_x$ tries to change the name of an object from Y to X and FGT$_y$ tries to delete that object using the old name Y or vice versa.

For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

FGT$_x$: renameObject(*College, Student, Mark, _, _, attribute, Grade*)
FGT$_y$: deleteObject(*College, Student, Mark, _, _, attribute*)

Applying FGT$_x$ first will change the name of the attribute *Mark* to a new name *Grade*. Then at the time of applying FGT$_y$ attribute *Mark* will not be found. Applying FGT$_y$ first will delete the attribute *Mark* from the class *Student*, then at the time of applying FGT$_x$ attribute *Mark* will not be found.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 18 and 19 between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to add a *relation* between two different objects in the system and FGT$_y$ tries to delete one of the two objects which participates in that relation or vice versa. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT$_x$: addRelation(*_, College, Teacher, viewSTMark, _, [], method, College, Student, Name, _, _, attribute, read*)
  FGT$_y$: deleteObject(*College, Student, Name , _, _, attribute*)

  Applying FGT$_x$ first will add the *read* relation between the method *Teacher.viewSTMark* and the attribute *Student.Name,* indicating that the method *viewSTMark* has *read* access to attribute *Name*. This means that the attribute *Student.Name* is now referenced from another object in the system which means that it could not be deleted by FGT$_y$. Conversely, applying FGT$_y$ first will delete the attribute *Student.Name* that will prohibit applying FGT$_x$ because at that time one of the participating objects in the *relation* will not be defined.

  These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 20-22 between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to add an object in a container and FGT$_y$ tries to delete that container or vice versa. An example would be if FGT$_x$ tries to add a member (attribute or method) in a class while FGT$_y$ tries to delete that class. Adding a member in a class will prohibit deleting that class. The same when FGT$_x$ tries to define a new parameter in a method *m* and FGT$_y$ tries to delete that method using the old signature of the method (the parameter list before adding the new parameter).

These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 23 is the same as conflicts 9-12 but with *association* relations. It occurs when one of the FGT tries to add a new *association* relation with label *X* between two classes while another FGT tries to change the label of another *association* relation that exists between the same two classes to a new label *X*. This means that the two classes will have two *association* relations with the same label which is prohibited. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT$_x$: renameRelation(*teach, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association, supervise*)

  FGT$_y$: addRelation(*supervise, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association*)

  Note that the reason for just considering *association* relations and excluding the other types of relations in this type of conflict is the following:

  c. For *read*, *write*, *call* and *type* relations: There is no label for these relations as explained in section 3.3 and also it is prohibited for two objects to have more than one *relation* of the same type (*read, write, call* or *type*).

  d. For *extends* relation it is also prohibited for two classes to have more than one *extends* relation between them at the same time.

  The type of this conflict is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 24 occurs when FGT$_x$ and FGT$_y$ try to change the label of two different *association* relations that exists between two specific objects to the same label. This means that the two associations will have the same label which is prohibited. The other types of relations are excluded from this conflict for the same reasons explained above.

  The type of this conflict is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 25 between FGT$_x$ and FGT$_y$ occurs when FGT$_x$ tries to change the label of a relation (*association or extends*) that exists between two classes from *X* to *Y* and FGT$_y$ tries to delete that relation using the old label *X* or vice versa. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

FGT$_x$: renameRelation(*teach, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association, supervise*)

FGT$_y$: deleteRelation(*teach, College,Teacher,_,_,_,class,College,Student,_,_,_, class, association*)

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 26 between FGT$_x$ and FGT$_y$ occurs when FGT$_x$ tries to change the access mode of specific object *A* from a less restricted mode to a more restricted one and FGT$_y$ tries to add a relation where the destination of the relation is the object *A* and the relation requires that the access mode of the object *A* should be the less restricted one. In the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

FGT$_x$: changeOAMode(*College, Student, Name , _, _, attribute, public, private*)

FGT$_y$: addRelation(*_, College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

Applying FGT$_x$ first will change the access mode of the attribute *Student.Name* from *public* to *private* which will prohibit applying FGT$_y$. Alternatively, applying FGT$_y$ first will add a *read* relation between the method *Teacher.viewStMark* and the attribute *Student.Name* this will prohibit applying FGT$_x$.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 27 between FGT$_x$ and FGT$_y$ occurs when the two FGTs try to add an *extends* relation between two classes where the subclass in the two FGTs is the same class *A*, this means that class *A* will have multiple superclass (multiple inheritance) which is not allowed.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 28 between FGT$_x$ and FGT$_y$ occurs when FGT$_x$ tries to add an *extends* relation between class *A* and *B* where *A* is the source (superclass) of the relation and *B* is the destination (subclass) of the relation. At the same time, FGT$_y$ tries to add an *extends* relation between class *A* and *B* where *B* is the source (superclass) of the relation and *A* is the destination (subclass) of the relation which is prohibited.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

8.3.2 Uni-Directional Conflict

A uni-directional conflict is a conflict that may occur between two FGTs, but only if they are applied to the system in a specific order. In the following a discussion of each type of uni-directional conflict as catalogued in Table 8.2 is given. Illustrative use of the simplified UML class diagram of a *College* system is continued. Note that all the uni-directional conflicts are ordering-conflicts. A tool should determine the specific order that should be followed to resolve the conflict between the two refactorings. In reference to the conflict-pairs in Table 8.2, if FGT$_y$ is applied first, then the conflict will be resolved.

Table 8.2: Uni-directional FGT conflict-pairs

No	FGT$_x$	FGT$_y$
A.	renameObject(P,C,M,PR,LT,OT,X)	addRelation(_,P,C,M,PR,LT,OT,_,_,_,_,_,_,_)
B.	renameObject(P,C,M,PR,LT,OT,X)	addRelation(_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_)
C.	renameObject(P,C,M,PR,LT,OT,X)	renameRelation(_,P,C,M,PR,LT,OT,_,_,_,_,_,_,_,_)
D.	renameObject(P,C,M,PR,LT,OT,X)	renameRelation(_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_,_)
E.	renameObject(P,C,M,PR,LT,OT,X)	deleteRelation(_,P,C,M,PR,LT,OT,_,_,_,_,_,_,_)
F.	renameObject(P,C,M,PR,LT,OT,X)	deleteRelation(_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_)
G. *	addObject(P1,C1,X,_,_,_,_,_,attribute)	renameObject(P2,C2,_,_,_,attribute,X)
H.*	addObject(P1,C1,X,_,_,_,_,_,method)	renameObject(P2,C2,_,_,_, method,X)
I.*	renameObject(P1,C1,_,_,_,attribute,X)	addObject(P2,C2,X,_,_,_,_,_,attribute)
J. *	renameObject(P1,C1,_,_,_,method,X)	addObject(P2,C2,X,_,_,_,_,_, method)
K.*	addObject(P1,C1,X,_,_,_,_,_,attribute)	addObject(P2,C2,X,_,_,_,_,_,attribute)
L.*	addObject(P1,C1,X,_,_,_,_,_,method)	addObject(P2,C2,X,_,_,_,_,_, method)

* Note: Assume *P1.C1* is one of the ancestor's of *P2.C2*.

- Conflicts A and B between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to change the name of object from *X* to *Y* and FGT$_y$ tries to add a *relation* between two objects where the object used in FGT$_x$ is the source or the destination object in the relation, and the relation continues to use the old name of the object *X*. Applying FGT$_x$ first then FGT$_y$ will cause a conflict but applying the two FGTs in a reverse order will not cause any conflicts. For example, in the following two FGTs in the *College* System will result in a conflict:

  FGT$_x$: renameObject(*College, Student , Name, _, _, attribute, StName*)

  FGT$_y$: addRelation(*_, College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

Applying FGT$_x$ first will change the name of the attribute in the class *Student* from *Name* to *StName* which will prohibit applying FGT$_y$ that uses the old name which is not defined at this point. However, applying FGT$_y$ first then FGT$_x$ will not cause any conflicts because at the time of applying FGT$_y$ the two objects used in the relation are defined. The ID of the two objects will be used to store the *relation* in the facts' database, as:

read(*RelID, _, viewStMarkID, NameID*)

Subsequently, FGT$_x$ may change the name of the attribute *Student.Name* to *Student.StName* but this will not cause a conflict at the level of the stored ID information.

- Conflicts C-F are similar to the conflicts in the previous point but instead of adding a new relation they change the label or delete an existing relation. In both of the cases, they use the old name of the object.

- Conflicts G and H between FGT$_x$ and FGT$_y$ occur when FGT$_x$ tries to add a new member with name *X* to the class *C1* and FGT$_y$ tries to change the name of another member in the class *C2* to the name *X* where class *C1* is an ancestor of class *C2*. Applying FGT$_x$ first will prohibit applying FGT$_y$ because FGT$_y$ in this case will try to redefine an inherited member which is not allowed. For example, a conflict will occur in the following two FGTs in the *College* System:

  FGT$_x$: addObject(*College, Student, Major, _, _, _, _, _, attribute*)
  FGT$_y$: renameObject(*College, PostGradStudent, ResTitle, _, _, attribute, Major*)

  Applying FGT$_x$ first will add the attribute *Major* to the *Student* class which then prohibits applying FGT$_y$ because the attribute *Major* is inherited from a superclass and it is not allowed to redefine it. Applying FGT$_y$ first will not cause any conflict because after changing the name in the subclass to *Major* there is no problem to add an attribute with the same name in a superclass.

- Conflicts I-L are almost the same as conflicts in the previous point. In these conflicts, one of the two FGTs tries to define a new member or rename an existing member in the subclass X where the new name is the same as one that is already defined by the second FGT in one of the class X's ancestors, which means that the inherited member from the superclass is being redefined and this is prohibited.

8.4 Conflict Algorithm

Because each refactoring is represented as a collection of FGTs ordered in FGT-DAGs it is possible to identify, at an FGT-level, where conflicts might occur in two refactorings. At a later stage, such conflicts will need to be resolved.

In this section, a conflict detection and resolving algorithm (**detectResolveConflict**) is defined. The algorithm is based on detecting and resolving conflicts of FGTs that constitute these refactorings. To do so, the algorithm uses the information given in Table 8.1 and 8.2, which is stored in the database of the tool as **fgtConflicts** facts.

Algorithm 8.1 gives the pseudo-code of the **detectResolveConflict** algorithm. The algorithm takes as input two refactorings X and Y. The algorithm initially assumes that X and Y can be applied to the system in either order. It ends with one of the following verdicts:

1. The (possibly user-modified) refactorings are conflict-free, and can be applied in any order.
2. The (possibly user-modified) refactorings are conflict-free provided they are applied in a specified order.
3. The refactorings are in conflict and the user has withdrawn one of them.

The algorithm works in a nested loop fashion. For each FGT-DAG$_i$ of refactoring X, the algorithm starting from the root, taking each FGT$_{ii}$ in FGT-DAG$_i$ and checking if there is a conflict between it and all the FGT$_{jj}$ in every FGT-DAG$_j$ of refactoring Y. The traversal of FGT-DAGs is also in a top-down fashion, starting at the root. Every pair (FGT$_{jj,}$ FGT$_{ii}$) or (FGT$_{ii,}$ FGT$_{jj}$) is checked for a match with the **fgtConflict** facts. If a match is found this means that there is a conflict between FGT$_{ii}$ and FGT$_{jj}$.

It is emphasized that the algorithm traverses the FGT-DAGs in the two refactorings under consideration from top to bottom. This allows one to detect the first occurrence of conflict in the two FGT-DAGs, perhaps to resolve this conflict and, if required, to modify the FGTs in the FGT-DAG$_i$ that are sequentially dependent on a modified FGT. This means that when the algorithm reaches FGT$_{ii}$ of FGT-DAG$_i$ of refactoring X, all the FGTs before FGT$_{ii}$ (ancestors of FGT$_{ii}$) in FGT-DAG$_{ii}$ are conflict-free (there are no conflicts between any of them with any other FGTs in refactoring Y). This is illustrated in Figure 8.6.

Algorithm 8.1 (Conflict detection & resolving algorithm)

```
detectResolveConflict( Ref X, Ref Y )
Input:    Ref X: a conflict-free redundancy-free set of FGT-DAGs of refactoring X
          Ref Y: a conflict-free redundancy-free set of FGT-DAGs of refactoring Y
Output:   Detect & Resolve conflicts between refactorings X & Y

For each $FGT_{ii}$ in $FGT\text{-}DAG_i$ (starting from the root) in Ref X do {
  For each $FGT_{jj}$ in $FGT\text{-}DAG_j$ (starting from the root) in Ref Y do {
    If there is a match between the pair ($FGT_{jj,}$ $FGT_{ii}$) and an fgtConflict fact then {
      switch (conflict-pair($FGT_{ii,}$ $FGT_{jj}$)) {
        ordering-conflict: { Determine the correct order of the two refactorings that
                             resolves the conflict between $FGT_{ii}$ and $FGT_{jj}$. If this order is
                             opposite to an order determined in a previous iteration of the
                             algorithm, behave as for a cancelling-conflict }
        removable-conflict: { Ask the developer to modify $FGT_{ii}$ to resolve the conflict,
                              accounting for all matters mentioned above. This includes
                              modifying, if necessary, FGTs in the sub-DAG rooted in
                              $FGT_{ii}$. If such modification is not allowed by the context, then
                              behave as for the cancelling-conflict }
        cancelling-conflict { Ask the developer to choose one of the refactorings X or Y.
                              Delete the chosen refactorings from the system.
                              End the detectResolveConflict procedure }
      } //end switch
    } //end If
  } //end for
} //end for
```

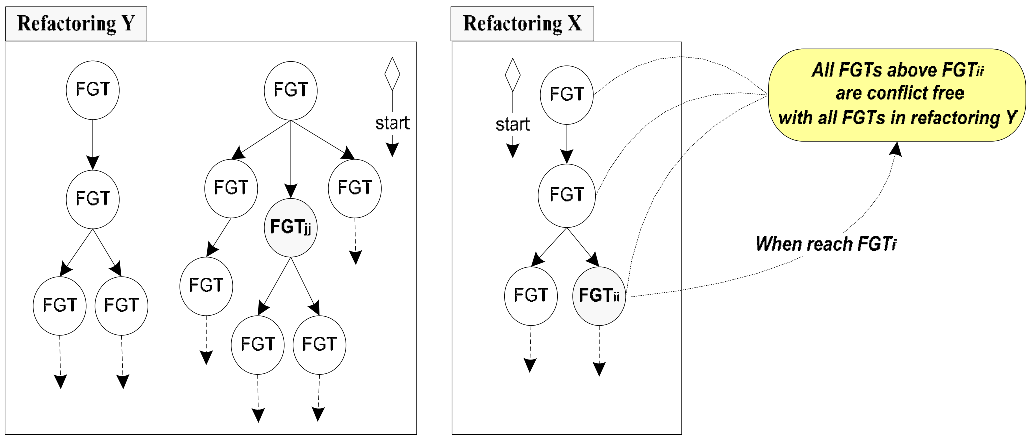

Refactoring Y
Refactoring X
FGT
FGTjj
FGTii
start
All FGTs above FGTii are conflict free with all FGTs in refactoring Y
When reach FGTi

Figure 8.6: Conflict detection & resolving algorithm

To resolve the conflict when it is detected, the algorithm determines the type of the conflict between the pair FGT$_{ii}$ and FGT$_{jj}$. Three different types of conflicts are identified:

1. If the conflict is an ordering-conflict, then the algorithm relies on the order of FGT$_{ii}$ and FGT$_{jj}$ in the matched **fgtConflict** fact to determine the correct order of the two refactorings. For example, in reference to the *College* system, suppose that FGT$_{ii}$ and FGT$_{jj}$ in Figure 8.6 are as the following:

FGT$_{ii}$: addRelation(_, *College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

FGT$_{jj}$: renameObject(*College, Student, Name, _, _, attribute, StName*)

In this example, **detectResolveConflict** algorithm will find a match between the pair (FGT$_{ii,}$ FGT$_{jj}$) and the fact

fgtConflict(renameObject(*P,C,M,PR,LT,OT,X*), addRelation(_,_,_,_,_,_,_,*P,C,M,PR,LT,OT*,_))

This conflict is explained in row B of Table 8.2. According to the information stored in the refactoring tool's database, the conflict is an ordering-conflict. To resolve this conflict, therefore, the refactoring that contains the FGT which matches with the second argument of the **fgtConflict** fact should be applied first. In the example, FGT$_{ii}$ (from refactoring X) matches with the second argument of the fact. This means that refactoring X should be applied first, and then refactoring Y. Using this scenario, when apply refactoring X, the *read* relation will be added between the two objects using the old name of the attribute *College.Student.Name*. Thereafter, by applying refactoring Y, the name of the attribute *College.Student.Name* will be changed to *College.Student.StName*. The conflict is resolved.

Note, however, that it is possible that at a later stage, another ordering-conflict is detected. If this conflict can only be resolved by an opposite ordering to the one already determined then there is a deadlock between the two refactorings. The only way to resolve the deadlock is to withdraw one of the refactorings, as in the case of a cancelling-conflict.

2. If the conflict is a removable-conflict, then the algorithm asks the developer to modify FGT_{ii} in a way that will resolve the conflicts between the two FGTs. The algorithm then will traverse $FGT\text{-}DAG_i$ from FGT_{ii} down and modify all FGTs that sequentially depends on FGT_{ii} to reflect the modification of FGT_{ii}. For example, related to the *College* system, suppose that FGT_{ii} and FGT_{jj} in Figure 8.6 are as the following:

FGT_{ii}: addObject(*College, PostGradStudent, ResField, _, _, type(basic, string, 0), public, _, attribute*)

FGT_{jj}: renameObject(*College, PostGradStudent, ResTitle, _, _, attribute, ResField*)

In this example, **detectResolveConflict** algorithm will find a match between the pair ($FGT_{ii,}$ FGT_{jj}) and the fact

fgtConflict(addObject(*P,C,X,_,_,_,_,_,attribute*), renameObject(*P,C,_,_,_,attribute,X*))

this means that there is a conflict between the two FGTs. This is because as a result of the two FGTs, the class *PostGradStudent* will have two attributes with the same name *ResField* ,which is a conflict. This conflict is explained in row 10 of Table 8.1. According to the information stored in the refactoring tool's database, the conflict is a removable-conflict. To resolve this conflict, the algorithm will ask the user to choose another name to be used for the attribute in FGT_{ii} instead of *ResField* (*ResSubject* for example). Then the tool will modify FGT_{ii} to be:

addObject(*College, PostGradStudent, **ResSubject**, _, _, type(basic, string, 0), public, _, attribute*)
instead of

addObject(*College, PostGradStudent, **ResField**, _, _, type(basic, string, 0), public, _, attribute*)

After that, the algorithm will traverse $FGT\text{-}DAG_i$ from the node FGT_{ii} down until it reaches the leaves, changing each occurrence of the attribute *College.PostGradStudent.ResTitle* to *College.PostGradStudent.ResSubject*. Note, however, that it is possible that the modification is not allowed by the context. Then the only way to resolve the deadlock is to withdraw one of the refactorings, as in the case of a cancelling -conflict.

3. If the conflict is a cancelling-conflict, then the algorithm will ask the developer to choose one of the refactorings X or Y to be cancelled. The algorithm will delete the chosen refactoring from the system and terminate the execution of the algorithm. For example, suppose that FGT$_{ii}$ and FGT$_{jj}$ in Figure 8.6 are as the following:

FGT$_{ii}$: addRelation(_,*College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

FGT$_{jj}$: DeleteObject(*College, Student, Name, _, _, attribute*)

In this example, FGT$_{ii}$ tries to add a *read* relation from the method *College.Teacher.viewStMark* to the attribute *College.Student.Name*. At the same time FGT$_{jj}$ tries to delete the attribute *College.Student.Name* which leads to a conflict. This conflict is explained in row 19 of Table 8.1. According to the information stored in the database of the refactoring tool, the conflict is one of the cancelling-conflicts. To resolve this conflict, the algorithm will ask the user to choose one of the two refactorings (X or Y) to be cancelled.

8.5 LAN Motivated Example

Consider the motivated LAN example. Assume a multi-user system, such that one user wants to apply the refactoring **pullUpMethod** to pull up the method *accept* from the subclasses *FileServer*, *PrintServer* to their superclass *Server*, and another user wants to move the *accept* method from the *FileServer* class to the *Packet* class. The latter user may be motivated, for example, by the fact that the *accept* method can directly access the variable *receiver* in the class *Packet*. Clearly that there is a conflict between the two refactorings **moveMethod** and **pullUpMethod** because there is no possibility to move the method *accept* from the *FileServer* to the class *Packet* and at the same time pull it up from the *FileServer* to the superclass *Server*.

In order to discover this fact algorithmically by the FGT-based tool, the **detectResolveConflict** procedure will be called and the two refactorings, shown in Figure 8.7, will be sent as parameters to the procedure. After executing the procedure, a match is found between the following pair of FGTs (FGT$_{ii}$, FGT$_{jj}$) and one of the **fgtConflict** facts:

FGT$_{ii}$: deleteObject(*Lan, FileServer, accept, _, [Packet], method*)

FGT$_{jj}$: deleteObject(*Lan, FileServer, accept, _, [Packet], method*)

where FGT$_{ii}$ is from refactoring **pullUpMethod** and FGT$_{jj}$ is from refactoring **moveMethod**. The **fgtConflict** fact that the match occurs with is:

fgtConflict(deleteObject(*P,C,M,PR,LT,ObjT*), deleteObject(*P,C, M,PR,LT,ObjT*))

This conflict is explained in row 5 of Table 8.1. According to the information stored in the refactoring tool's database, the conflict is one of the cancelling-conflicts. To resolve this conflict, the algorithm will require that one of the two refactorings (**moveMethod** or **pullUpMethod**) be cancelled.

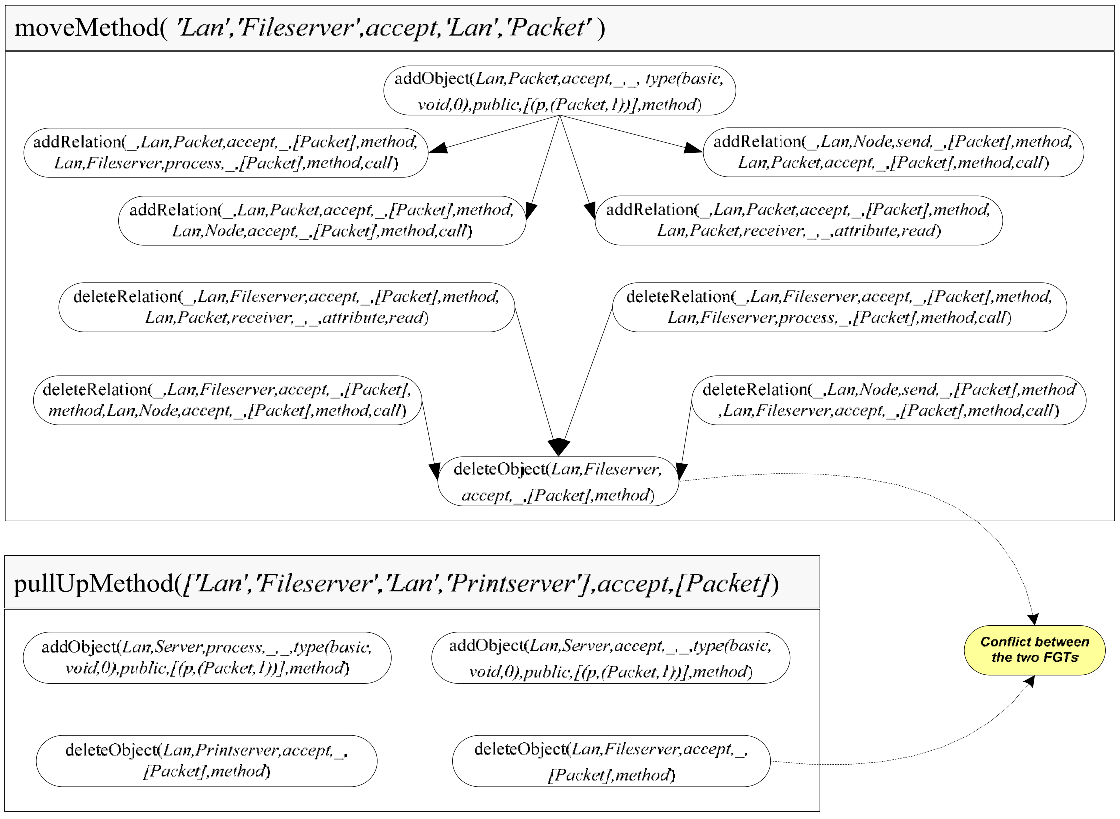

moveMethod('Lan','Fileserver',accept,'Lan','Packet')
addObject(Lan,Packet,accept,_,_, type(basic, void,0),public,[(p,(Packet,1))],method)
addRelation(_,Lan,Packet,accept,_,[Packet],method, Lan,Fileserver,process,_,[Packet],method,call)
addRelation(_,Lan,Node,send,_,[Packet],method, Lan,Packet,accept,_,[Packet],method,call)
addRelation(_,Lan,Packet,accept,_,[Packet],method, Lan,Node,accept,_,[Packet],method,call)
addRelation(_,Lan,Packet,accept,_,[Packet],method, Lan,Packet,receiver,_,_,attribute,read)
deleteRelation(_,Lan,Fileserver,accept,_,[Packet],method, Lan,Packet,receiver,_,_,attribute,read)
deleteRelation(_,Lan,Fileserver,accept,_,[Packet],method, Lan,Fileserver,process,_,[Packet],method,call)
deleteRelation(_,Lan,Fileserver,accept,_,[Packet], method,Lan,Node,accept,_,[Packet],method,call)
deleteRelation(_,Lan,Node,send,_,[Packet],method ,Lan,Fileserver,accept,_,[Packet],method,call)
deleteObject(Lan,Fileserver, accept,_,[Packet],method)
pullUpMethod(['Lan','Fileserver','Lan','Printserver'],accept,[Packet])
addObject(Lan,Server,process,_,_,type(basic, void,0),public,[(p,(Packet,1))],method)
addObject(Lan,Server,accept,_,_,type(basic, void,0),public,[(p,(Packet,1))],method)
deleteObject(Lan,Printserver,accept,_, [Packet],method)
deleteObject(Lan,Fileserver,accept,_, [Packet],method)
Conflict between the two FGTs

Figure 8.7: Conflicts between refactorings moveMethod & pullUpMethod

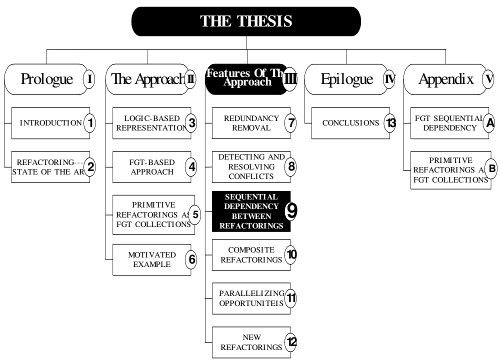
8.6 Reflections on Conflicts

The conflicts discussed in this chapter should not be confused with sequential dependencies—neither at the refactoring-level nor at the FGT-level. These conflicts deal with what may not happen before applying a refactoring (or FGT). For example, Table 8.2 (entries A-E) specify that an object may not be renamed before changing (i.e. adding, deleting, or renaming) a relation associated with the object. This does not mean that a relation involving an object has to be changed before renaming the object. What has to happen before apply some refactoring or FGT is, broadly speaking, a sequential dependency issue, and this will be discussed in the next chapter.

Chapter 9

SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS

9.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of Th Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS A FGT COLLECTIONS B

As in the previous chapter, the concern is with two arbitrary refactorings, R_i and R_j. However, in this case, it is assumed that R_i and R_j are conflict-free. The first question to consider is the following: Is it possible to apply the two refactorings in any order on any system that satisfies the preconditions of both? The answer is, of course, that it is possible, since the refactorings are assumed to be conflict-free. A second question then, is whether it could be possible to apply R_j after applying R_i in a system that initially satisfied R_i's precondition, but not R_j's. There is one of two possible answers:

1. Yes, it would be possible, provided that the initial state of the system is such that it satisfies those precondition conjuncts of R_j that are not realized as a result of applying R_i first.

2. No, it is not possible, because there is some inherent contradiction between the pre- and postconditions of R_i and R_j—even though they are conflict-free. In this case, the deadlock problem arises.

It should be emphasized the assumption of conflict freedom between R_i and R_j is, initially, strict—i.e. it is assumed that there is no ordering-conflict that can be resolved if one refactoring is applied after the other. In considering the deadlock problem in section 9.5, however, this restriction will be lifted.

Refactoring R_j is sequentially dependent on refactoring R_i ($R_i \rightarrow R_j$) if some or all of the precondition conjuncts of refactoring R_j are satisfied by applying refactoring R_i first. Thus, the definition of refactoring sequential dependency is similar to the definition of FGT sequential dependency, namely:

Refactoring R_j sequentially depends on refactoring R_i if and only if the postcondition conjuncts of refactoring R_i satisfies one or more precondition conjuncts of refactoring R_j.

As an example, suppose that a system has a package with name *P* and that a user intends to apply two refactorings R_i and R_j. Assume—as shown in Figure 9.1—that the two refactorings involve the following respective transformations on the system: Refactoring R_i adds a new class *A* in the package *P*; and refactoring R_j adds a new attribute *Attn* in the class *P.A* which is added in refactoring R_j. Clearly, refactoring R_j sequentially depends on refactoring R_i.

A batch of refactorings may be produced in a multi-developer environment in which groups of developers work on the same system. Potentially, such a batch of refactorings may be large with many sequential dependencies between the different refactorings. It would therefore be useful to have an automated way of discovering such sequential dependencies between refactorings.

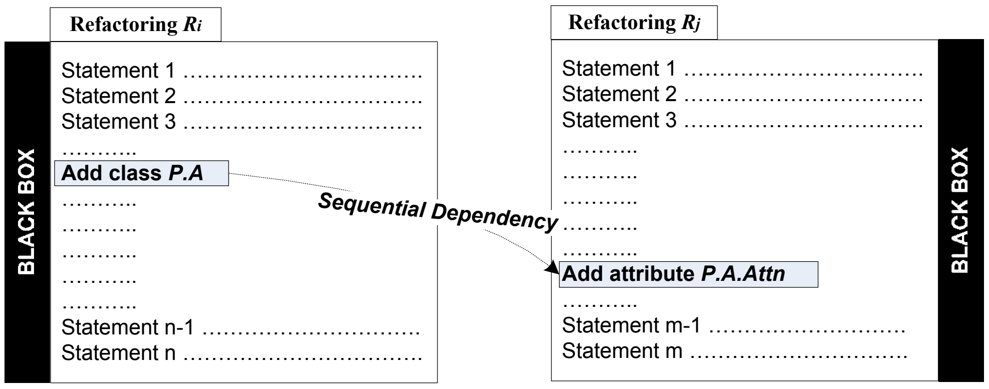

Refactoring Ri
BLACK BOX
Statement 1 ....
Statement 2 ....
Statement 3 ....
Add class P.A
Statement n-1 ....
Statement n ....
Sequential Dependency
Refactoring Rj
Statement 1 ....
Statement 2 ....
Statement 3 ....
Add attribute P.A.Attn
Statement m-1 ....
Statement m ....
BLACK BOX

Figure 9.1: Sequential dependency between refactorings R_i & R_j

9.2 Sequential Dependency in Previous Approaches

A straightforward approach to apply a set of refactorings in a batch to a system is simply to traverse the batch to find a candidate whose precondition conjuncts are satisfied by the system and then to apply it to the system. Then search the batch again, looking for a new candidate and so on until the list is finished or none of the remaining refactorings can be applied. Each time the new refactoring's precondition conjuncts will be checked against the system under consideration, which means that, potentially, the tool has to make many references to the system. Note that in practice, the description of the system may be very large, and references to such a large system therefore runs the risk of becoming costly.

Such an approach suffers from the classical disadvantages of "greedy" algorithms: non-optimal behaviour. For example, in the example presented in Figure 9.1, if the tool by chance chooses refactoring R_j first, then its precondition conjuncts will not be satisfied. Refactoring R_i will then be chosen, its precondition conjuncts checked, and processing then proceeds (This means that R_i will be applied to the system). Subsequently, the tool will go back to refactoring R_j to check its precondition conjuncts again, which implies duplication of work and effort in referencing the underlying system.

In order to solve the above problem, various authors have proposed alternative approaches to find sequential dependencies between refactorings by trying to find such relations without a need to check the underlying system under consideration, thus reducing the time needed to reference the underlying system. In [39, 52, 55, and 70], an approach is proposed that infers sequential dependency relations between the different refactorings by comparing their pre- and postcondition conjuncts without a need to reference the state of the system under consideration.

In the example presented in Figure 9.1, the existence of class *A* in package *P* is one of the conjuncts in the precondition of refactoring R_j. Also one of the conjuncts of the postcondition of refactoring R_i is precisely the existence of class *A* in package *P*. Because the postcondition of refactoring R_i satisfies a conjunct of the precondition conjuncts of refactoring $R_{j,}$ a tool can infer that R_j sequentially depends on R_i.

While depending on pre- and postcondition conjuncts of refactorings does indeed establish whether or not there is a sequential dependency between two refactorings, the following should be noted:

1. Sometimes it is impossible to infer the sequential dependency between the two refactorings by considering the pre- and postcondition conjuncts in isolation. Figure 9.2 shows an example of two refactorings *Ri* and R_j where part of the two refactorings does the following respective transformations on the system: Refactoring R_i adds a new class *A* in the package *P*; and refactoring R_j changes the name of the class *P.A* to another name *P.C* as indicated in the example.

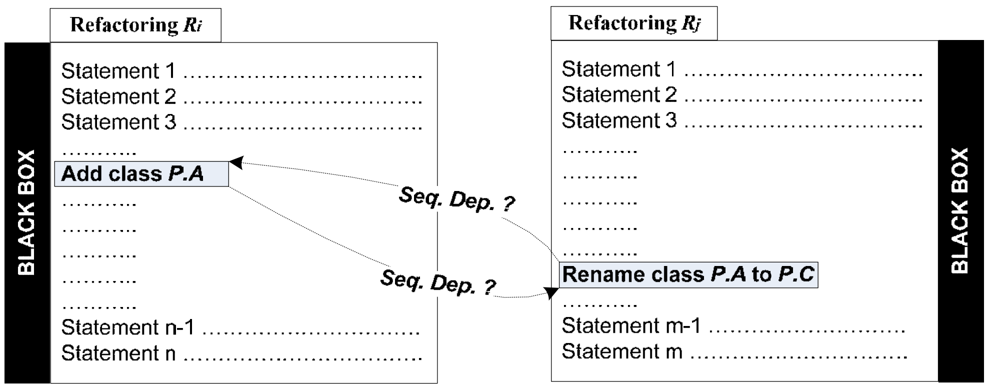


Figure 9.2: Ambiguous sequential dependency

Considering just these transformations of the two refactorings the following pre- and postcondition of the two refactorings can be distinguished:

Precondition of R_i: class *A* not defined in package *P*

Postcondition of R_i: class *A* is defined in package *P*

Precondition of R_j: class *A* is defined in package *P*, class *C* is not defined in package *P*

Postcondition of R_j: class *A* is not defined in package *P*, class *C* is defined in package *P*

From the information that can be inferred from the previous pre- and postcondition conjuncts of the two refactorings, two scenarios can be distinguished:

- It is clear that part of the postcondition conjuncts of refactoring R_i is included in the precondition conjuncts of refactoring R_j. From this, the tool can infer that R_j is sequentially dependent on R_i ($R_i \rightarrow R_j$).

- It is also clear that part of the postcondition conjuncts of refactoring R_j is included in the precondition conjuncts of refactoring R_i. From this, the tool can infer that R_i is sequentially dependent on R_j ($R_j \rightarrow R_i$).

Therefore, there is an ambiguity about the sequential dependency relation between the two refactorings if they are viewed in isolation of the underlying system. As far as can be established, authors of approaches such as in [39, 52, 55, and 70] do not address this problem. The ambiguity may be resolved by checking the underlying system to discover the real state of the system. This makes it possible to choose the correct sequential dependency relation that is imposed by the system under consideration. In the previous example, if the class *P.A* is already defined in the system then scenario **b** is the correct scenario where ($R_j \rightarrow R_i$). On the other hand, if class *P.A* is not defined in the system then scenario **a** is the correct scenario where ($R_i \rightarrow R_j$).

2. Since refactoring tools to date typically implement each refactoring as black-boxed sequence of coding statements, it is not possible to establish at what specific points in the respective code blocks the two refactorings become sequentially dependent. This leads to the following shortcomings:

a. It is not possible to exploit any possibilities for implementing the refactorings in parallel.

b. It is not possible to exploit any possibilities for removing redundancy between the different refactorings.

Of course, this latter problem does not arise when the refactorings happen to correspond to FGTs. However, primitive refactorings may, in general, include many actions (FGTs). This is true also of composite refactorings.

9.3 Sequential Dependency between FGT-Based Refactorings

Finding sequential dependencies between refactorings can be based on finding sequential dependencies at the level of the FGTs that constitute the refactorings. As shown in Figure 9.3, representing refactorings R_i and R_j—discussed in the previous example—as collections of FGTs ordered in FGT-DAGs means that the sequential dependency between every pair of FGTs in the two refactorings can be checked. The figure shows that there is a sequential dependency between the two FGTs:

FGT$_{ii}$: addObject(*P, A, _, _, _, _, public, _, class*)
FGT$_{jj}$: addObject(*P, A, Attn, _, _, type(basic, int, 0), public, _, attribute*)

where FGT$_{jj}$ sequentially depends on FGT$_{ii}$. Finding just one case of sequential dependency between a pair of FGTs in the two refactorings is sufficient to establish the sequential dependency between the two refactorings. In the example, the fact that refactoring R_j sequentially depends on refactoring R_i, means that refactoring R_i needs to be applied first.

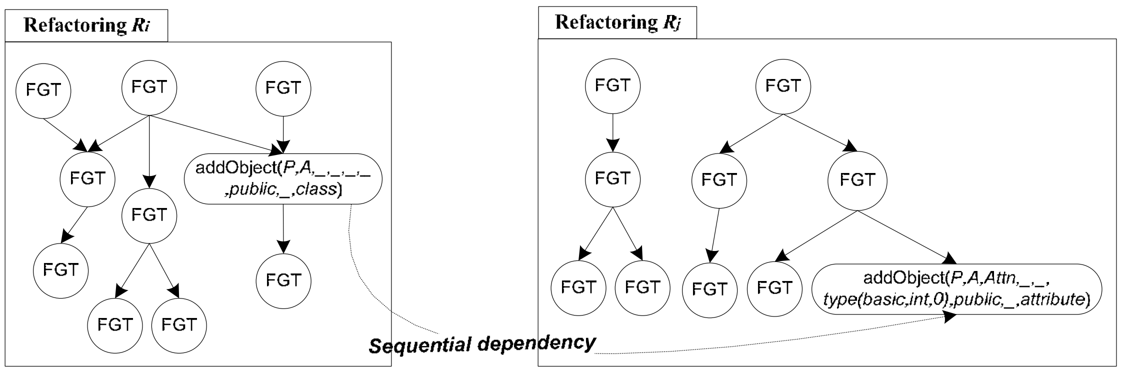

Refactoring Ri
Refactoring Rj
FGT
addObject(P,A,_,_,_,_ ,public,_,class)
addObject(P,A,Attn,_,_, type(basic,int,0),public,_,attribute)
Sequential dependency

Figure 9.3: Sequential dependency in FGT-based approach

To do that, the various possibilities of sequential dependency that may occur between the different FGTs have to be examined. Recall that these have been pre-catalogued as shown in Figure 4.1 and explained in more details in Appendixes A.1 and A.2. As explained in sections 4.3.2 and 4.3.3, two categories of sequential dependencies between FGTs were identified: Uni-directional FGT sequential dependencies and Bi-directional FGT sequential dependencies.

Note that, one of the advantages of distinguishing between two types of FGT sequential dependencies is that the information can be used to solve the ambiguity problem discussed in the previous section with respect to refactorings. If there is a bi-directional sequential dependency between two FGTs that appear within two respective refactorings, this means that there is ambiguity between the two refactorings as well. In this case, the underlying system should be referenced in order to establish which refactoring should be applied first and which one second in a given context, or whether, in fact, there is a deadlock problem, as discussed in section 9.5 below.

Thus, by identifying the type FGTs involved in two refactorings (uni-directional or bi-directional), the tool can determine whether or not there is a need to reference the underlying system (i.e. in the case of bi-directional sequential dependency). This is the basis for the sequential dependency algorithm, discussed in the next section.

9.4 Sequential Dependency Algorithm

Representing refactorings as a collection of FGT-DAGs allows one to establish exactly at which part of the two refactorings the sequential dependency between the two occurs. In this section, an algorithm (**sequentialDependency**) to find the sequential dependency between two refactorings is defined. The algorithm is base on finding the sequential dependency at the level of FGTs which constitute these refactorings. It uses the **uniDirSD** and **biDirSD** facts already stored in the database of the tool.

As mentioned before, access on the underlying system is a time-consuming process if the system is large. Therefore, the algorithm has been designed to minimize such access. This criterion is taken into consideration in the proposed **sequentialDependency** algorithm by working in three phases:

Phase one: In this phase, the algorithm tries to find the sequential dependency relations between the two refactorings using the **uniDirSD** facts. It takes each FGT from the first refactoring (FGT_{ii}) and checks if it has a **uniDirSD** with any other FGTs of the second refactoring (FGT_{jj}). Finding a single match is enough for the algorithm to determine the sequential dependency between the two refactorings. Note that in this phase the algorithm takes a decision without having to access the underlying system. If the algorithm does not find any of the **uniDirSD** between any pair of FGTs in the two refactorings then the algorithm goes to the next phase.

Phase two: In this phase, the sequential dependency algorithm tries to find the sequential dependency relation between the two refactorings using the **biDirSD** facts. It takes each FGT from the first refactoring (FGT_{ii}) and checks if it has a **biDirSD** with any other FGTs of the second refactoring (FGT_{jj}). If one is found, then the algorithm has to check the underlying system to resolve the ambiguity. If the algorithm does not find any of the **biDirSD** between any pair of FGTs in the two refactorings then the algorithm goes to the next phase.

Phase three: In this phase, the algorithm checks the refactoring-level pre- and postcondition conjuncts of the two refactorings to infer the sequential dependency between the two refactorings. Note that the approach used here is the same as the approach described in the second part of section 9.2 with a major difference: the concern here is just with refactoring-level pre- and postcondition conjuncts and not with the entire set of refactoring pre- and postcondition conjuncts.

It should be noted that the **sequentialDependency** algorithm given below can be used to establish the sequential dependency relationship between any two refactorings that are represented as FGT-DAGs. It is, however, a requirement that the refactorings be both conflict-free and deadlock-free. The matter of deadlock freedom is taken up in section 9.5 below, while the previous chapter has shown how conflict freedom can be established. Three different kinds of conflicts between pairs of refactorings were mentioned: ordering-conflicts (where conflict can be resolved by ordering one of the refactorings before the other); cancelling-conflicts (where conflict can only be resolved by withdrawing one of the refactorings); and removable-conflicts (where conflicts can be resolved by appropriately modifying FGTs that comprise one of the refactorings).

In general, the **sequentialDependency** algorithm can be used to establish the sequential dependency relationships between appropriately selected refactorings in a *batch* of refactorings. The outcome is then one or more **refactoring directed acyclic graph (REF-DAGs)**, as illustrated in Figure 9.4. Each node in a REF-DAG represents *one* of the refactorings and contains the FGT-DAGs of that refactoring. When the **sequentialDependency** algorithm finds that refactoring Y is sequentially dependent on refactoring X, then node X becomes the father of node Y in one of the REF-DAGs. If there is no sequential dependency between two different REF-DAGs then they can be processed and applied in parallel.

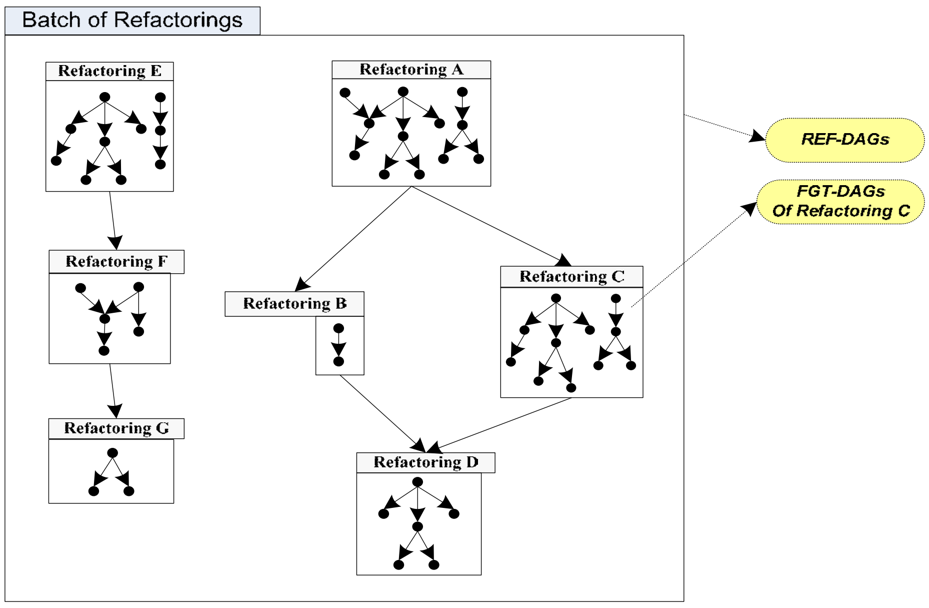


Figure 9.4: Refactoring Directed Acyclic Graphs (REF-DAGs)

Algorithm 9.1 (Sequential dependency algorithm)

```
sequentialDependency( Ref X, Ref Y )
Input:      Ref X: a conflict-free redundancy-free set of FGT-DAGs of refactoring X
            Ref Y: a conflict-free redundancy-free set of FGT-DAGs of refactoring Y
Assumption:   Ref X & Ref Y are deadlock-free
Output:     An indication that X→Y; or that Y→X; or that there is no sequential dependency
             relationship between X and Y

// Start of phase one
For each $FGT_{ii}$ in $FGT\text{-}DAG_i$ (starting from the root) in X do {
  For each $FGT_{jj}$ in $FGT\text{-}DAG_j$ (starting from the root) in Y do {
    Search unDirSD facts to find a match with the pair ($FGT_{ii}$, $FGT_{jj}$) or the pair ($FGT_{jj}$, $FGT_{ii}$)
    If there is a match then {
                        Determine sequential dependency between X & Y
                        Return result } //end If
  } //end for
} //end for
// End of phase one. Start of phase two
For each $FGT_{ii}$ in $FGT\text{-}DAG_i$ (starting from the root) in X do {
  For each $FGT_{jj}$ in $FGT\text{-}DAG_j$ (starting from the root) in Y do {
    Search biDirSD facts to find a match with the pair ($FGT_{ii}$, $FGT_{jj}$) or the pair ($FGT_{jj}$, $FGT_{ii}$)
    If there is a match then {
                        Check the underlying system to determine the direction of the
                        sequential dependency
                        Return result } //end If
  } //end for
} //end for
// End of phase two. Start of phase three
{ Check pre- and postcondition conjuncts at the refactoring-level of the two refactorings,
  Return result }
// End of phase three
```

Algorithm 9.1 gives the pseudo-code for the **sequentialDependency** algorithm. The algorithm takes as input two refactorings X and Y. It then works in a nested loop fashion. For each FGT-DAG$_i$ of refactoring X, the algorithm starts from the root and takes each FGT$_{ii}$ in FGT-DAG$_i$.

It checks if there is a sequential dependency between it and each FGT_{jj} in every $FGT\text{-}DAG_j$ of refactoring Y, in each case also starting from the root of $FGT\text{-}DAG_j$. For every pair, of (FGT_{jj}, FGT_{ii}) or (FGT_{ii}, FGT_{jj}), the algorithm checks if there is a match with the **uniDirSD** facts. When a match is found this means that there is a sequential dependency between FGT_{ii} and FGT_{jj}, which means a sequential dependency between the two refactorings X and Y. Then the loop breaks and the algorithm returns the result to the calling procedure.

If the nested loop completes without finding any match, then the algorithm goes to the next phase by starting the nested loop again but this time searching for a match with the **biDirSD** facts. If the nested loop completes without finding any match, then the algorithm goes to the next phase to check the refactoring-level pre- and postcondition conjuncts of the two refactorings.

9.5 Deadlock Problem

A deadlock between two refactorings occurs when each one of the two refactorings sequentially depends on the other. In other words, each one of the two refactorings needs the other one to be applied to the system, to satisfy its precondition conjuncts. As a result, none of them can be applied to the system.

The idea is explained in Figure 9.5. The FGTs in the two refactorings X and Y have the following sequential dependency relations:

FGTx1 $\rightarrow$ FGTy1 (*This means that Refactoring Y is sequentially dependent on Refactoring X*)
FGTy2 $\rightarrow$ FGTx2 (*This means that Refactoring X is sequentially dependent on Refactoring Y*)

Because the sequential dependencies between the two refactorings go in both directions, it can be concluded that there is a deadlock situation.

Note that in all the algorithms presented in the previous chapters of the thesis, an assumption of deadlock freedom between the different refactorings is considered. The same assumption is also made in the rest of the thesis.

It should be noted that deadlocks can only arise if an irrational attempt is made to refactor an existing system. If users refer to the current system only, their requested refactorings will not result in deadlock with each other, even if they request refactorings on the system independently of one another. Any requested refactoring has to rely on the state of the system to satisfy its precondition conjuncts, or on applying some other prior refactoring to the system first. In the latter case, the precondition conjuncts of the latter system have to be satisfied by the system's state, etc. A deadlock can only occur if a user mistakenly attempts to satisfy the precondition conjuncts of one refactoring by specifying another, whose precondition depends on the first. One way in which this could happen, for example, is if end users are given the ability to build their own refactorings—as discussed in chapter 12—and this result in unorganised dummy refactorings which have deadlock between each other.

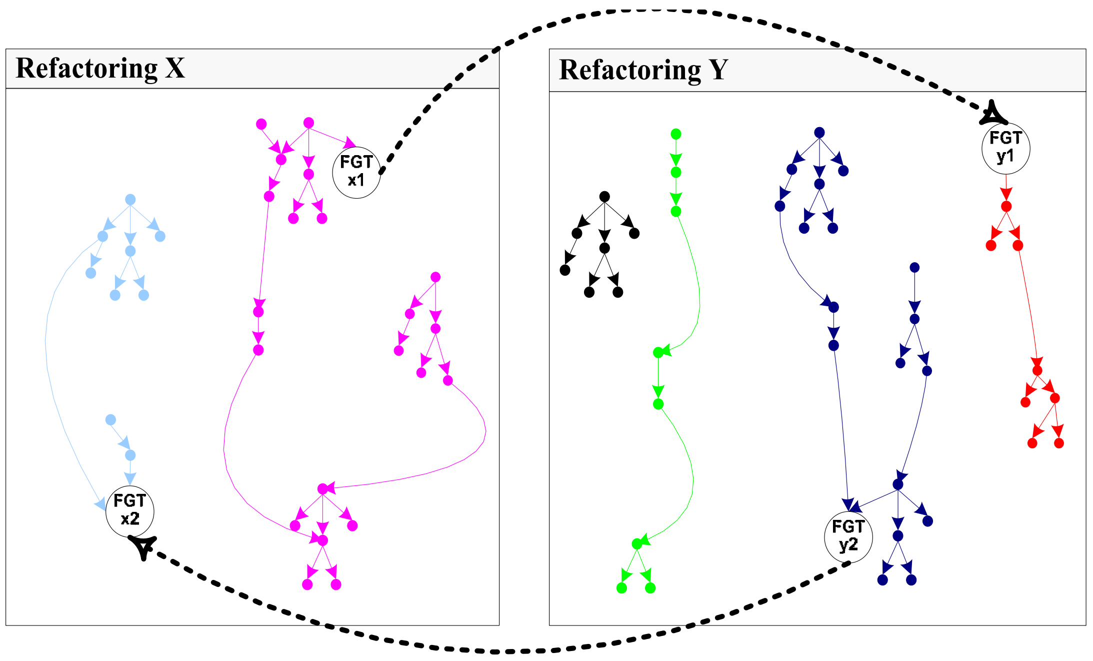


Figure 9.5: Deadlock problem

In order to detect the deadlock between two refactorings, **deadLockDetection** algorithm may be used. Algorithm 9.2 provides the pseudo-code for the deadLockDetection algorithm. The algorithm takes as input two refactorings X and Y. It then searches to find sequential dependencies between each pair of FGTs in the two refactorings. The algorithm works in the three phase manner as in the **sequentialDependency** algorithm described in section 9.4, with the following main difference:

When the algorithm finds the first sequential dependency between a pair of FGTs, it stores this sequential dependency relation and continues with the remaining of FGTs, searching for all other sequential dependency relations. Each time a new sequential dependency relation is discovered, it is checked with the stored one (the first discovered one). If it is in the opposite direction this means that there is a deadlock, the algorithm is terminated and a "DeadLock" message is returned to the calling procedure; otherwise the algorithm will continue until all FGT pairs have been checked.

If no deadlock is discovered then the algorithm will start with the third phase. It will check the pre- and postcondition conjuncts at the refactoring-level to infer if there is a deadlock between the two refactorings or not. For this, the algorithm uses the following rule:

If (refactoring-level precondition conjuncts of refactoring X contains some of the postcondition conjuncts of refactoring Y) and (refactoring-level precondition conjuncts of refactoring Y contains some of the postcondition conjuncts of refactoring X), then there is a deadlock between the two refactorings.

In addition, the algorithm will check the sequential dependencies discovered in this phase with the stored one (if any) from the previous two phases.

Note that the assumption to date has been that X and Y are conflict-free in a strict sense—i.e. there is no ordering-conflict between X and Y. The deadLockDetection algorithm can be modified in an obvious way to detect possible deadlock between X and Y if this restriction is lifted. It would simply be a matter of noting the direction of the ordering-conflict at the start, and declaring a deadlock between X and Y if a sequential dependency is later discovered in the opposite direction.

Algorithm 9.2 (Deadlock detection algorithm)

```
deadLockDetection( Ref X, Ref Y )
Input:    Ref X: a conflict-free set of FGT-DAGs of refactoring X
          Ref Y: a conflict-free set of FGT-DAGs of refactoring Y
Output:   An indication of whether or not X & Y are deadlocked

Let SDFound = false
// Start of phase one
For each $FGT_{ii}$ in $FGT\text{-}DAG_i$ (starting from the root) in X do {
  For each $FGT_{jj}$ in $FGT\text{-}DAG_j$ (starting from the root) in Y do {
    Search unDirSD facts to find a match with the pair ($FGT_{ii}$, $FGT_{jj}$) or the pair ($FGT_{jj}$, $FGT_{ii}$)
    If there is a match then determine the sequential dependency ($X \rightarrow Y$) or ($Y \rightarrow X$)
    If (! SDFound) then {store the SD relation, SDFound=true}
      Else {compare the new SD relation with the stored one
             If it is in the opposite direction then return "DeadLock"}
  } //end for
} //end for
// End of phase one. Start of phase two
For each $FGT_{ii}$ in $FGT\text{-}DAG_i$ (starting from the root) in X do {
  For each $FGT_{jj}$ in $FGT\text{-}DAG_j$ (starting from the root) in Y do {
    Search biDirSD facts to find a match with the pair ($FGT_{ii}$, $FGT_{jj}$) or the pair ($FGT_{jj}$, $FGT_{ii}$)
    If there is a match then check the underlying system to determine the direction of the SD
    If (!SDFound) then {store the SD relation, SDFound=true}
      Else {compare the new SD relation with the stored one
             If it is in the opposite direction then return "DeadLock"}
  } //end for
} //end for
// End of phase two.  Start of phase three
Check pre- and postcondition conjuncts at the refactoring-level of the two refactorings
Determine the sequential dependencies between X & Y
Compare the discovered SDs with each other and with the stored one (if there is)
If there are two SDs in the opposite directions then return "DeadLock"
Return "DeadLock-Free"
// End of phase three
```

9.6 LAN Motivated Example

Consider the motivating example given in chapter 6. To find the sequential dependency between the three proposed refactorings (**pullUpMethod**, **createClass** and **encapsulateAttribute**), the sequential dependency algorithm will take FGT-DAGs of two refactorings each time to check if there is a sequential dependency between them. As a result of executing the algorithm, while checking the two refactorings **pullUpMethod** and **createClass** during phase one, the algorithm finds that FGT:

addObject(Lan, Server, accept, _, _, type(basic, void, 0), public, [(p, type(complex, Packet, 0))], method)

in refactoring **pullUpMethod** is sequentially dependent on FGT:

addObject(*Lan, Server, _, _, _, _, public, _, class*)

in refactoring **createClass**. As a result, the sequential dependency algorithm indicates that refactoring **pullUPMethod** is sequentially dependent on refactoring **createClass.** The resulting REF-DAG will be as shown in Figure 9.6.

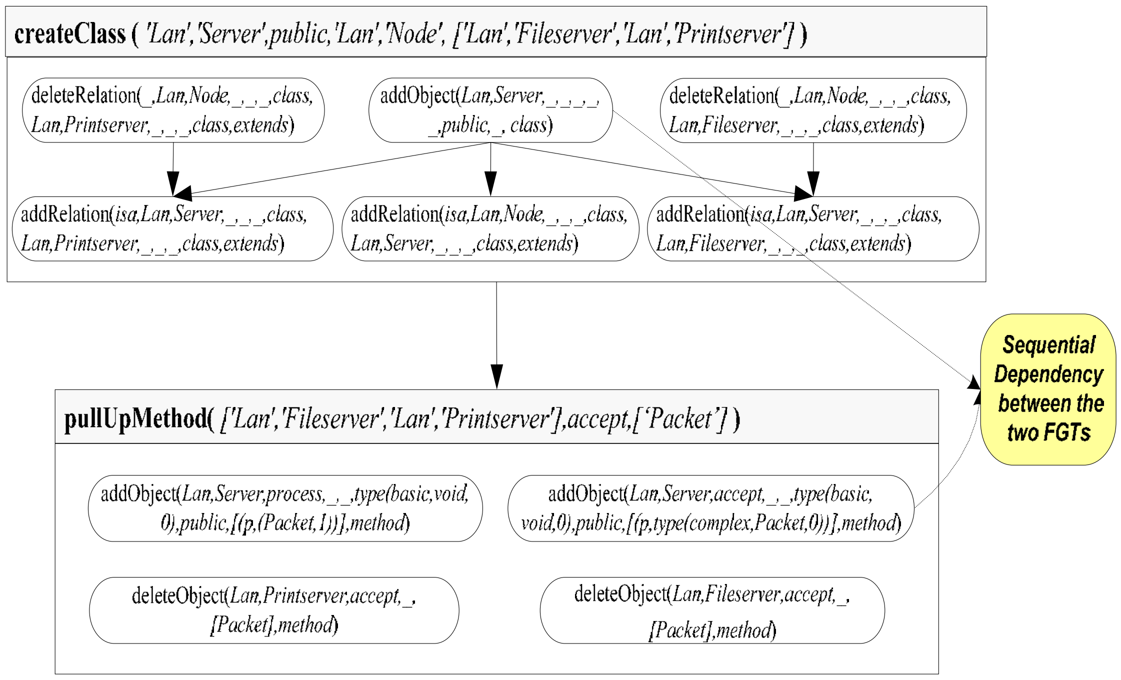
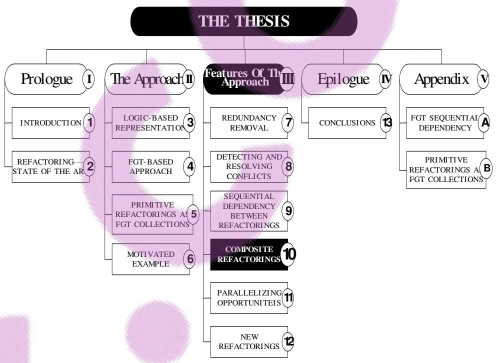

createClass ('Lan','Server',public,'Lan','Node', ['Lan','Fileserver','Lan','Printserver'])
deleteRelation(_,Lan,Node,_,_,_,class, Lan,Printserver,_,_,_,class,extends)
addObject(Lan,Server,_,_,_,_, _,public,_, class)
deleteRelation(_,Lan,Node,_,_,_,class, Lan,Fileserver,_,_,_,class,extends)
addRelation(isa,Lan,Server,_,_,_,class, Lan,Printserver,_,_,_,class,extends)
addRelation(isa,Lan,Node,_,_,_,class, Lan,Server,_,_,_,class,extends)
addRelation(isa,Lan,Server,_,_,_,class, Lan,Fileserver,_,_,_,class,extends)
Sequential Dependency between the two FGTs
pullUpMethod(['Lan','Fileserver','Lan','Printserver'],accept,['Packet'])
addObject(Lan,Server,process,_,_,type(basic,void, 0),public,[(p,(Packet,1))],method)
addObject(Lan,Server,accept,_,_,type(basic, void,0),public,[(p,type(complex,Packet,0))],method)
deleteObject(Lan,Printserver,accept,_, [Packet],method)
deleteObject(Lan,Fileserver,accept,_, [Packet],method)

Figure 9.6: Sequential dependency between refactorings createClass & pullUpMethod

Chapter 10

COMPOSITE REFACTORINGS

10.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING---STATE OF THE AR 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

A developer who restructures a system, starts with some design goals in mind. In practice, it is likely that a single primitive refactoring will not meet the design goal in isolation. Instead, it may be necessary to jointly group primitive refactorings into a "batch" [38, 70], which is then applied to the model as one unit. Such a batch of primitive refactorings that addresses one or more of a developer's design goals is termed a composite refactoring. Of course, the composite refactoring created in this way could subsequently be combined with others, thus creating new ones, and so on [49].

Composite refactoring are conventionally specified as a *sequence* of primitive refactorings, the assumption being that they will be applied in that specific order. However, a necessary (but not sufficient) condition for successfully applying such a sequence to a system is that it should respect the so-called *sequential dependencies* between the constituent primitive refactorings. Briefly, if one or more precondition conjuncts of refactoring P are logically entailed by the postcondition of refactoring Q, then P is sequentially dependent on Q, denoted by Q→P.

For the purposes of the present discussion, it will be assumed that the primitive refactorings in a composite have been rationally selected—i.e. that there are no conflicts between the various primitive refactorings. (The previous chapters have also shown how such conflicts may be detected.) In order to illustrate relevant concepts, consider seven primitive refactorings, A…G, that are to be used in composite refactoring X. Suppose that they have the following sequential dependencies:

$$A \rightarrow B,\ A \rightarrow C,\ E \rightarrow F,\ F \rightarrow G,\ B \rightarrow D,\ C \rightarrow D$$

Assume, too, that the user has specified X as the following refactoring sequence:

$$X = \langle A, E, F, B, C, G, D \rangle$$

Note that this sequence respects the sequential dependencies: any refactoring that is sequentially dependent on another, will be processed after the latter. (As an aside, note that X embodies the following mutually independent refactoring subsequences:

$$<A\rightarrow B,\ A\rightarrow C,\ B\rightarrow D,\ C\rightarrow D> \text{ and } <E\rightarrow F,\ F\rightarrow G>$$

In principle, these subsequences may be processed in parallel.)

The straightforward approach to apply such a composite to some system, is to focus on the primitive refactorings' pre- and postconditions in isolation, as shown in Figure 10.1. To applying the composite, the precondition of the first primitive refactoring, A, is checked against the system's state. If it is satisfied then refactoring A is applied. The precondition of the next primitive refactoring, E, is checked against the system, it is applied, and so on.

Dealing with composite refactorings in this way is vulnerable to the rollback problem: if, at some point, a precondition of one of the primitive refactorings in the composite is not satisfied, then the refactoring tool has to rollback all the primitive refactorings in the composite that had previously been applied to the system, so as to restore the system to its original state.

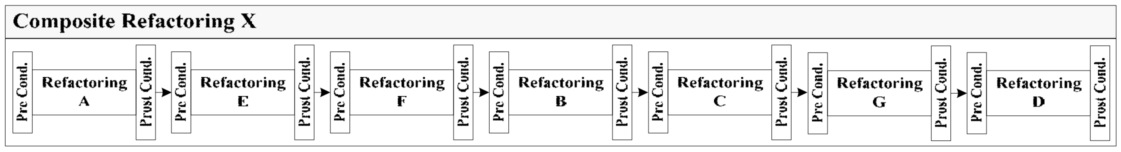


Figure 10.1: Straightforward approach

In [35, 38, 52, and 70], the concept of a composite precondition / postcondition, as illustrated in Figure 10.2, was proposed to deal with this rollback problem. The idea is to derive the *composite's* precondition and postcondition by considering the pre- and postconditions of its individual primitive refactorings. Note that the derived composite precondition conjuncts are not simply the conjunction of all precondition conjuncts of its constituent primitive refactorings. Doing so would neglect the transformations performed between the evaluation of the different conditions. For instance, assume that a composite X consists of two primitive refactorings (R_1 and R_2) where R_2 sequentially depends on R_1. Suppose the precondition of R_1 is P_1^P_2 and the precondition of R_2 is P_3^P_4. Suppose, also, that the postcondition of R_1 is P_3 (or, more generally, that it logically implies P_3, but not P_4). Then the precondition of the composite X is P_1^P_2^P_4. The same also applies for deriving the postcondition of the composite.

At the time of refactoring, the set of composite precondition conjuncts is checked against the system state before applying any primitive refactoring in the sequence to the system. If these are satisfied, then the primitive refactorings may be applied to the system in the sequence given, or indeed in any sequence that respects the sequential dependencies between primitive refactorings. Under such circumstances, no rollback will be necessary.

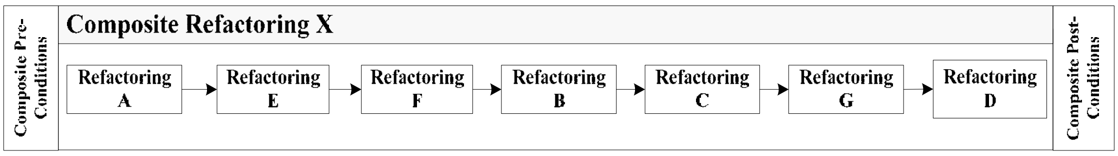


Figure 10.2: Composite refactoring in composite precondition approaches

The following section considers the implications of composite refactorings in the context of the FGT paradigm proposed in this thesis.

10.2 FGT-based Composite Refactoring

Recall that chapter 5 has catalogued the commonly mentioned primitive refactorings, together with their associated preconditions. That chapter also indicated how a given primitive refactoring that is to be applied on some system can be expressed as an equivalent FGT-list. As a consequence, an FGT-based procedure for composing several primitive refactorings, stored as FGT-lists, may seem obvious. Simply select the desired sequence of primitive refactorings that are to form a composite refactoring and place them in a list that respects their sequential dependency.

At this point, the composite-level's pre- and postcondition can be computed from the refactoring-level pre- and postconditions of all the refactorings inside the composite, exactly in the same way as described by previous authors. However, to determine whether or not the composite can be applied to the system without rollback, it is now no longer sufficient merely to check the composite-level precondition. In addition, the FGT-enabling preconditions of FGT-DAGs in the composite should also be checked against the system before deciding whether or not to apply the composite.

The fact that the FGT-enabling preconditions have to be computed means that some additional work has to be undertaken when using FGTs, if rollback is to be avoided. However, there are potential gains to be had from this cost, but to exploit them; the FGT-lists of individual primitive refactorings need to be decomposed into equivalent FGT-DAGs as described in chapter 4. Under these circumstances, it is at least of theoretical interest to explore whether the FGT-DAGs can be merged further. The practical value of doing this will be considered at the end of the chapter.

The procedure **compositeRefactoring**, to be described below, merges the FGT-DAGs of primitive refactorings, and determines the composite- and FGT-enabling preconditions. The procedure assumes that the following holds:

a. All the refactorings (whether or not they are primitive refactorings) that the developer wants to include in the composite are redundancy-free and conflict-free.

b. The refactorings are in a sequence whose order respects the sequential dependencies between them.

In order to build the composite refactoring X described in the previous section, the procedure **compositeRefactoring** is called as follows:

compositeRefactoring([*Refactoring A, Refactoring E, Refactoring F, Refactoring B,*
Refactoring C, Refactoring G, Refactoring D])

The procedure then executes the steps given below, which will be explained in more detail later:

Step 1: Generate the system-specific FGT-list corresponding to each primitive refactoring in the composite.

Step 2: Build a set of FGT-DAGs for each FGT-list.

Step 3: Determine the sequential dependency relationship between every pair of FGTs in the composite refactoring.

Step 4: Use refactoring-level pre- and postconditions to infer (a) possible undetected sequential dependencies between refactorings; and (b) composite-level pre- and postconditions.

Step 5: Remove all redundancies from the FGT-DAGs.

Step 6: Determine the FGT-enabling precondition of each FGT-DAG in the composite.

Clearly, the first two steps can be carried out by using the primitive refactoring procedures described in chapter 5 (in step 1); and the **build-FGT-DAG** algorithm found in section 4.3.4 (in step 2).

In the third step, it is obviously unnecessary to check dependency relationships between FGTs within each primitive refactoring's set of FGT-DAGs, since these are already reflected by the arcs in the FGT-DAGs. To check the dependency relationships between FGTs in different FGT-DAG sets, an adapted form of the algorithm in section 9.4 (to determine the sequential dependency between two refactorings) can be used. Recall that this algorithm operates on a pair of sets of FGT-DAGs and that it terminates upon finding a sequential dependency between a single pair of FGTs.

The first adaptation is to find *all* the sequential dependencies between FGTs in the two refactorings, and *not* just the first one. Under the assumption that the original list of primitive refactorings were rationally assembled, there will not be any circular paths (i.e. conflicts) generated by this process.

The second adaptation is that, when dealing with **biDirSD**, it is unnecessary to access the underlying system to determine the direction of the sequential dependency. Instead, the direction may be inferred from the order of primitive refactorings in the original list that is provided as input to the **compositeRefactoring** procedure.

Step 3 therefore, involves the following on each pair of primitive refactorings, say S and T, represented respectively as FGT-DAG sets: Consider all FGT pairs comprising of an FGT in S and an FGT in T. Use the facts **uniDirSD** and **biDirSD** to determine whether or not they are sequentially dependent, and if so, connect them by an appropriate sequential dependency arc.

The outcome of step 3 is schematically shown by the dashed arrows in Figure 10.3(a), connecting various FGTs across different refactorings.

The bold arrows between refactorings in Figure 10.3(a) indicate sequential dependencies between them. In this particular example, this may be directly inferred from the fact that sequential dependencies had been established in step 3 between one or more their constituent FGTs.

However, in general, it may be the case that there is no such FGT-level sequential dependency, but nevertheless, a sequential dependency that is related to the possibility that one (or more) refactoring-level postcondition establishes one (or more) refactoring-level precondition conjuncts of another[1]. For example, the **pullUpAttribute** primitive refactoring requires, as a refactoring-level precondition, that the relevant attribute's access mode should not be *private* in any of the relevant subclasses. It may or may not be the case that the original sequence of primitive refactorings contains a primitive refactoring(s) to change everywhere the attribute's access mode to *public* or *protected*. If there are such primitive refactorings that change the access mode, then **pullUpAttribute** is sequentially dependent on them. If there are no such primitive refactorings, then the requirement that the attribute should not have a *private* access mode in any of the relevant subclasses, becomes a composite-level pre-requisite.

It is the task of step 4 to compare the refactoring-level pre- and postconditions of a primitive refactoring pair, S and T, to infer which of the pre- and postcondition conjuncts should serve as composite-level pre- and postconditions respectively, and which of them indicate a refactoring-level sequential dependency that was not established in step 3. Note that in the latter case, an application of the composite refactoring must ensure that the application of the various FGTs respects such refactoring-level sequential dependencies. Figure 10.3(b) shows the combined effect of steps 3 and 4: a new set of FGT-DAGs made up of the original sets of FGT-DAGs, together with their associated composite-level pre- and postconditions.

However, before applying the composite, step 5 should be executed to remove possible redundancies that have arisen as a result of combining FGT-DAGs in the previous steps. The **reduction** algorithm described in section 7.4 may be used for this purpose.

Note that in step 6, the determination of the FGT-enabling precondition of each FGT-DAG in the composite, necessarily has to take place after reduction. This is to account for actual FGTs that are to be used in the composite, rather than those that were directly implied by the FGT-list that had been derived from the input primitive refactoring list.

[1] Note that the matter is, in fact, somewhat more subtle. It is also possible that an FGT postcondition establishes part of a refactoring level's precondition; or that a refactoring-level postcondition establishes part of an FGT's precondition. Example 2 below will give an illustration of the first of these possibilities.

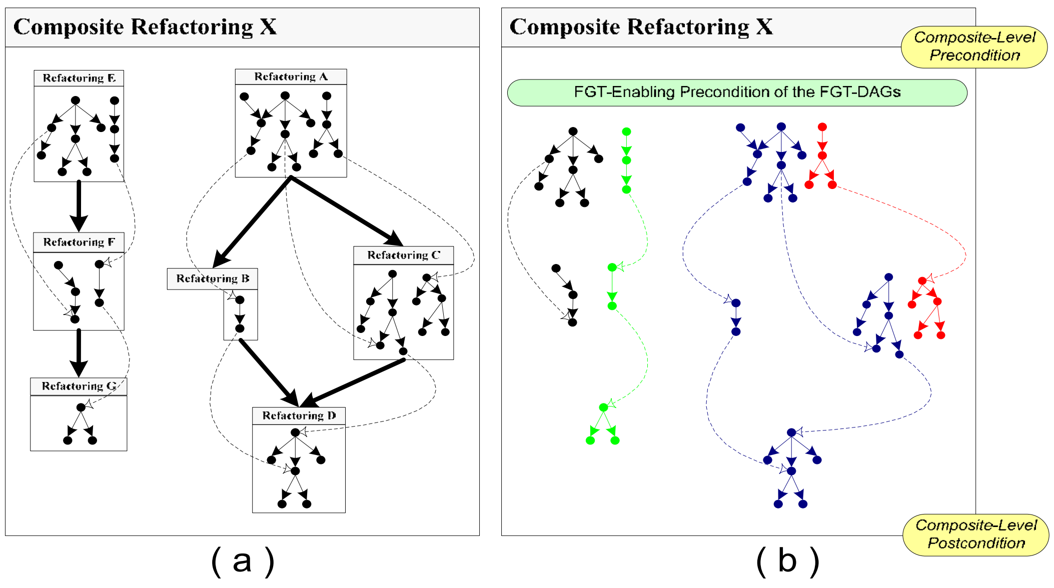


Figure 10.3: Composite refactoring in FGT approach

In summary, then, the **compositeRefactoring** procedure generates (a) a set of independent redundancy-free FGT-DAGs that reflects the actual transformations needed to achieve the composite refactoring; (b) a set of composite-level pre- and postconditions; and (c) the FGT-enabling precondition of each FGT-DAG in the composite.

At the time of refactoring, the composite refactoring is executed in two phases. In the first phase, two levels of preconditions are checked: (a) the composite-level preconditions, and (b) the FGT-enabling preconditions of the various FGT-DAGs.

If all relevant preconditions are satisfied then, in a second phase, the different FGTs are applied to the underlying system, again here, in the same order as they appear in the different FGT-DAGs. Processing the composite in two phases solves the rollback problem because the tool will not apply any FGT in the composite before checking that system to be refactored complies with the preconditions at the composite-level and FGT-enabling level.

The processed composite refactoring is like any other refactoring. All the operations that the approach offers for dealing with refactorings can also be carried out on composite refactorings (reduction, conflict detection & resolving, sequential dependency, and parallelization).

10.3 Examples

10.3.1 encapsulateAttribute Composite Refactoring

To illustrate the FGT approach for dealing with composite refactorings, the composite refactoring **encapsulateAttribute**—which is used to prevent direct accesses to a specific attribute—will be given as an example.

Figure 10.4(a) gives a UML class diagram for a simplified *College* system. The system has a package called *College* with three classes *Teacher*, *Student* and *Registration*. Note that the information extracted from the class diagram alone is not sufficient for refactoring. For example, if a method *m* is to be deleted from the class diagram using the primitive refactoring *deleteMethod*, then that method should be not referenced by any other object elements in the class diagram, and this kind of referencing information is not in the UML class diagram. The underlying logic representation of the class diagram should include this kind of extra information. To get such information, we have to refer to the code-level implementation of the system. Figure 10.4(a) shows such information represented as dashed arrows between the different object elements of the class diagram.

Suppose that one of the suggested enhancement to the class diagram of the *College* system is to encapsulate the attribute *Mark* in the *Student* class. This refactoring is useful for increasing modularity, by avoiding direct accesses of the local state of a *Student*. For this restructuring, the composite refactoring **encapsulateAttribute** can be constructed.

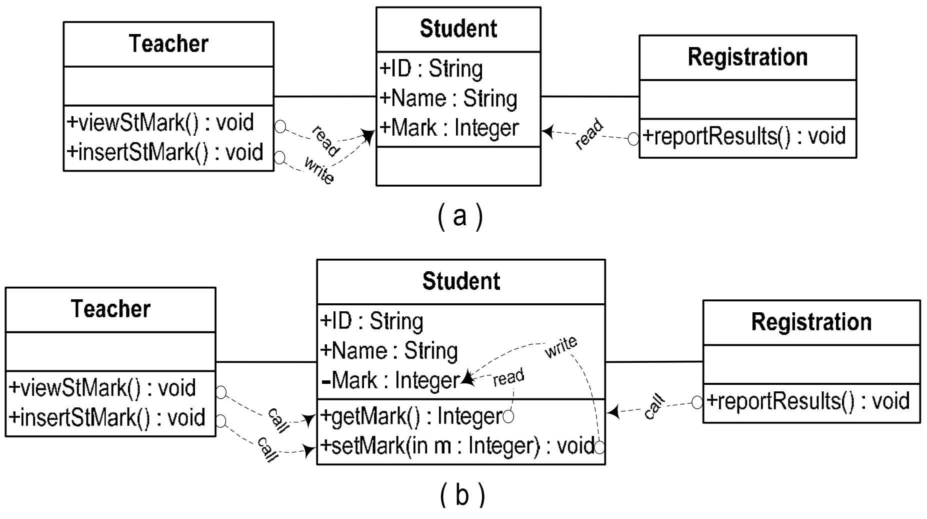


Figure 10.4: A simplified UML class diagram of a *college* system. (a) before and (b) after refactoring

The order of the primitive refactorings in the composite is shown in Figure 10.5. Note that the order reflects the sequential dependency that exist between the different refactorings in the composite. According to the order, a refactoring tool should first add the getter and setter methods. Then it should redirect the destination of all the *read/write* accesses from the attribute to them. After this stage, the attribute is not referenced by any object in the system. Therefore, the refactoring tool can change the access mode of the attribute from *public* to *private*.

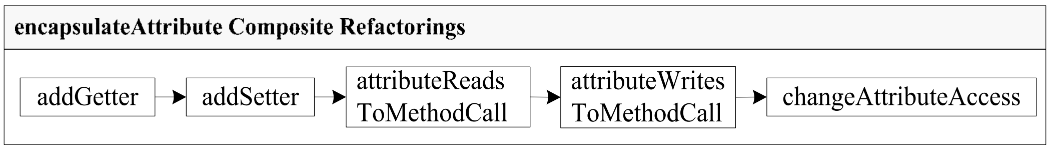

encapsulateAttribute Composite Refactorings
addGetter
addSetter
attributeReads ToMethodCall
attributeWrites ToMethodCall
changeAttributeAccess

Figure 10.5: encapsulateAttribute composite refactoring

In the refactoring tool, in order to encapsulate the attribute *College.Student.Mark*, the procedure:

compositeRefactoring([*addGetter, addSetter, attributeReadsToMethodCall,*
attributeWritesToMethodCall, changeAttributeAccess])

is used, where the arguments in the procedure refer to the primitive refactorings that are included in the composite **encapsulateAttribute.** (Note that, for conciseness, the arguments above are given in an abbreviated above. Their full form—as they should actually appear in the procedure call—is given in the middle column of Table 10.1.) As discussed above, the procedure will produce an FGT-list which represents the transformation actions to be performed as part of the encapsulation process. This FGT-list is shown in the right-hand column of Table 10.1.

The FGT-lists produced by each primitive refactoring in the composite are then allocated to one or more FGT-DAGs. Thereafter, the sequential dependencies between the different FGTs in the different primitive refactorings are found. These are shown as dashed arrows in Figure 10.6(a).

The solid arrows show the sequential dependencies between primitive refactorings that can be inferred from the dashed arrows. Note that the fact that there are only solid arrows between refactoring primitives whose FGTs show some sequential dependencies (i.e. the dashed arrows) is an indication that in this particular example, no primitive refactoring dependencies are induced by considering refactoring-level pre- and postconditions in step 4 of the

compositeRefactoring procedure. In fact, in this example, no composite-level preconditions are to be found.

Table 10.1: encapsulateAttribute refactoring

Comp-osite Ref.	**Sequence Of Primitive Refactorings**	**Sequence Of FGTs For Each Primitive Refactoring**
encapsulateAttribute(*'College', 'Student', 'Mark'*)	**addGetter**(*'College', 'Student', 'Mark'*)	**FGT1:** addObject(*College, Student, getMark, _, _, type(basic,int,0), public,[], method*) **FGT2:** addRelation(*_,College,Student,getMark,_,[],method,College, Student,Mark, _,_, attribute, read*)
	addSetter(*'College', 'Student', 'Mark'*)	**FGT3:** addObject(*College,Student,setMark,_,_, type(basic,void,0), public, [(p, type(basic,int,0))], method*) **FGT4:** addRelation(*_,College,Student,setMark,_,[int], method, College, Student, Mark,_,_, attribute, write*)
	attributeReadsToMethod-Call(*'College', 'Student', 'Mark', 'College', 'Student',getMark, []*)	**FGT5:** deleteRelation(*_,College,Teacher,viewStMark,_,[], method, College, Student, Mark,_,_, attribute,read*) **FGT6:** deleteRelation(*_,College,Registration, reportResults, _,[], method, College, Student, Mark,_,_,attribute, read*) **FGT7:**addRelation(*_,College,Teacher,viewStMark,_,[],method, College,Student,getMark,_,[],method,call*) **FGT8:** addRelation(*_,College,Registration,reportResults,_,[],method, College, Student, getMark,_,[], method, call*)
	attributeWritesToMethod-Call(*'College', 'Student', 'Mark', 'College', 'Student', setMark, [int]*)	**FGT9:** deleteRelation(*_,College,Teacher,insertStMark,_,[],method, College,Student, Mark,_,_,attribute,write*) **FGT10:** addRelation(*_,College,Teacher,insertStMark,_,[],method, College,Teacher, setMark, _,[int], method, write*)
	changeAttributeAccess(*'College', 'Student', 'Mark', private*)	**FGT11:** changeOAMode(*College,Student,Mark,_,_,attribute, public, private*)

Since there are no reductions to be made in step 5 of the **compositeRefactoring** procedure, Figure 10.6(b) shows the final result. It clearly indicates which FGTs may be applied independently to the system. For example, FGTs 1, 3, 5, 6 and 9 may be launched independently, while FGT 11 can only be applied once FGTs 5, 6 and 9 have completed.

Step 6 requires that the FGT-enabling precondition of each FGT-DAG in the composite has to be determined. Since there is no composite-level precondition in this example, the system should comply with the FGT-enabling precondition. If it does so, then the composite may be directly applied to the system without further checking of preconditions.

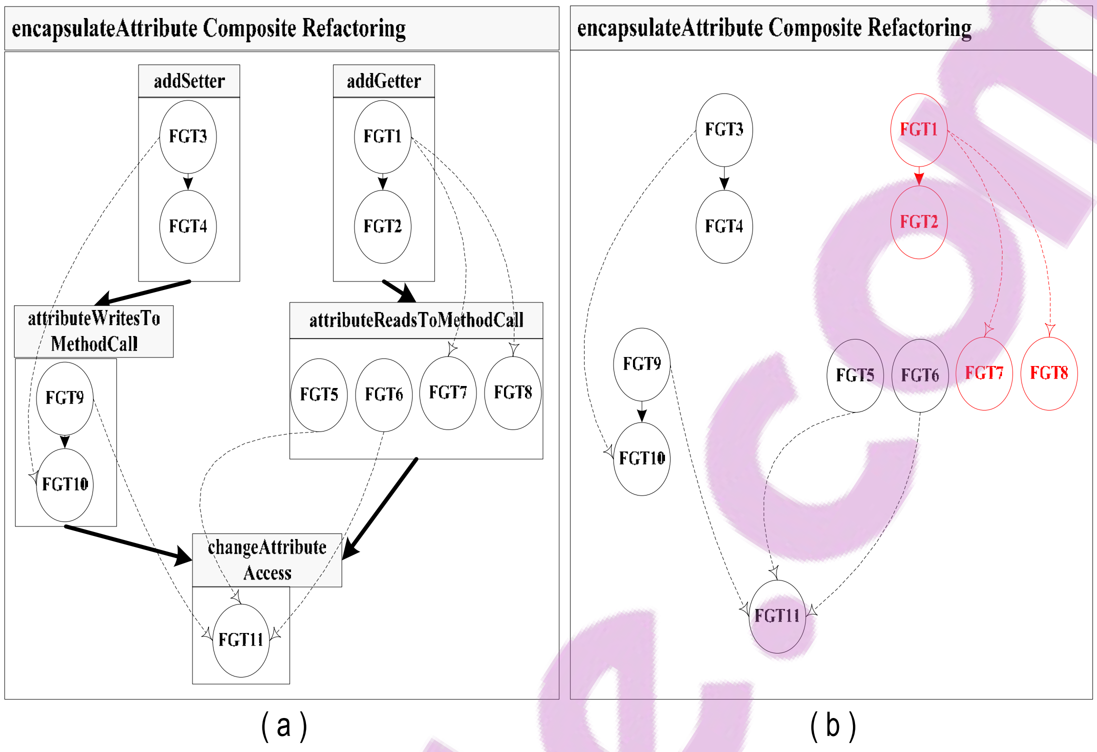

encapsulateAttribute Composite Refactoring
addSetter
addGetter
FGT1
FGT2
FGT3
FGT4
attributeWritesTo MethodCall
attributeReadsToMethodCall
FGT5
FGT6
FGT7
FGT8
FGT9
FGT10
changeAttribute Access
FGT11
encapsulateAttribute Composite Refactoring
(a)
(b)

Figure 10.6: encapsulateAttribute composite refactoring in FGT approach

10.3.2 enh-pullUpAttribute Composite Refactoring

A second example is provided to illustrate two aspects relating to step 4 that were not illustrated in the previous example: the way in which the composite-level precondition is derived; and the way in which sequential dependencies arise when FGT postconditions establish refactoring-level preconditions.

Suppose that the developer wants to create a new composite refactoring called **enh-pullUpAttribute**. The aim of the composite is to pull up an attribute from a set of subclasses to their common superclass. One of the precondition conjuncts of the primitive refactoring **pullUpAttribute**, described in section 5.3.2.8, is that the access mode of the attribute in all the subclasses where it is defined should not be *private*. The new proposed composite refactoring in this example solves this problem by changing the access mode of the attribute from *private* to *protected* which will give the ability to pull up it. The composite also defines a getter and a setter method for the pulled up attribute after pulling it up to the superclass. The composite refactoring **enh-pullUpAttribute** consists of the following actions:

1. Change the access mode of the attribute from *private* to *protected* in all the subclasses where the access mode of the attribute is *private*. This is done by using the primitive refactoring **changeAttributeAccess.**

2. Pull up the attribute from all the subclasses where it defined to their common superclass. This is done by using the primitive refactoring **pullUpAttribute**

3. Add getter and setter methods for the attribute that is now located in the superclass. This is done by using the primitive refactorings **addGetter** and **addSetter**.

Figure 10.7(a) gives a UML class diagram for a simplified system. The system has a package *P* with three classes *A*, *B* and *C*. *C* is the superclass of *A* and *B*. class *A* has an attribute *x* with *private* access mode. Class *B* has an attribute *x* with *protected* access mode.

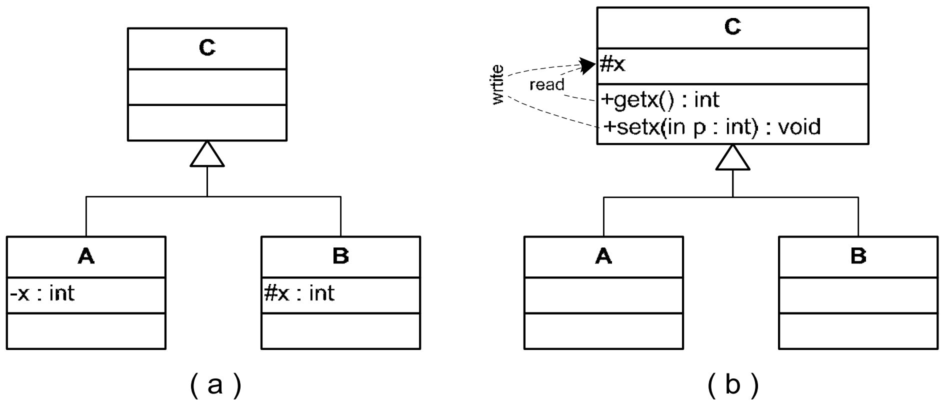


Figure 10.7: A simplified UML class diagram. (a) before and (b) after refactoring

Suppose that the developer wants to pull up the attribute *x* from the subclasses *A* and *B* to the superclass *C*. For this restructuring the composite refactoring **enh-pullUpAttribute** can be constructed. Note that, for this case, the primitive refactoring **pullUpAttribute** cannot be used directly because the access mode of the attribute *A.x* is *private* which means that the precondition of the refactoring will be not satisfied.

In the refactoring tool, in order to pull up the attribute *x*, the procedure:

compositeRefactoring([changeAttributeAccess(*'P', 'A', x, protected*), pullUpAttribute(*'P', 'C', x*), addGetter(*'P', 'C', x*), addSetter(*'P', 'C', x*)])

is used, where the arguments in the procedure refer to the primitive refactorings that are included in the composite **enh-pullUpAttribute**. The procedure will produce an FGT-list which represents the transformation actions to be performed as part of the pull up process. This FGT-list is shown in the right-hand column of Table 10.2.

Table 10.2: enh-pullUpAttribute refactoring

Comp-osite Ref.	Sequence Of Primitive Refactorings	Sequence Of FGTs For Each Primitive Refactoring
enh-pullUpAttribute (*'P'*, *'C'*, *x*)	changeAttributeAccess (*'P'*, *'A'*, *x*, *protected*)	**FGT1**: changeOAMode(*P,A,x,_,_,attribute, private, protected*)
	pullUpAttribute (*'P'*, *'C'*, *x*)	**FGT2**: addObject(*P,C,x,_,_, type(basic,int,0),protected,_,attribute*) **FGT3**: deleteObject(*P,A, x,_,_, attribute*) **FGT4**: deleteObject(*P,B, x,_,_, attribute*)
	addGetter(*'P'*, *'C'*, *x*)	**FGT5:** addObject(*P,C,getx,_,_,type(basic,int,0), public,[ ],method*) **FGT6:** addRelation(*_,P,C,getx,_,[ ],method,P,C,x,_,_,attribute,read*)
	addSetter(*'P'*, *'C'*, *x*)	**FGT7:** addObject(*P,C,setx,_,_,type(basic,int,0), public,[(p, type(basic, int,0))], method*) **FGT8:** addRelation(*_,P,C,setx,_,[int], method,P,C,x,_,_,attribute,write*)

The FGT-lists produced by each primitive refactoring in the composite are then allocated to one or more FGT-DAGs. Thereafter, the sequential dependencies between the different FGTs in the different primitive refactorings are found. These are shown as dashed arrows in Figure 10.8(a). The sequential dependencies between the primitive **pullUpAttribute**, **addGetter** and **addSetter** are inferred from these dashed arrows and represented as solid arrows in Figure 10.8(a).

In step 4 of the **compositeRefactoring** procedure, the refactoring-level pre- and postconditions are used to infer (a) possible undetected sequential dependencies between refactorings; (b) composite-level pre- and postconditions. In this example, the primitive refactoring **pullUpAttribute** inside the composite **enh-pullUpAttribute** has the following refactoring-level precondition conjuncts:

1. The attribute to be pulled up has the same type definition in all the subclasses in which it is defined.

2. The attribute to be pulled up does not have access mode *private* in any of the subclasses of the superclass

Let P.A denotes class A in package P, P.B denotes class B in package P, etc. Then, when **pullUpAttribute**'s refactoring-level precondition conjuncts are instantiated in terms of the objects in the system to be refactored, they become:

1. The attribute *x* has type *int* in both P.A and P.B.

2. The access mode of *x* in P.A is not *private*.

3. The access mode of *x* in P.B is not *private*.

Since the conjunct in line 2 is satisfied by the postcondition conjuncts of FGT1 in Table 10.2, the two primitive refactorings **changeAttributeAccess** and **pullUpAttribute** are sequentially dependent. This relation is represented by the solid arrow between the two refactorings in Figure 10.8(a).

The composite-level precondition conjuncts are those that remain after removing the conjunct in line 2, namely:

1. The attribute *x* has type *int* in both P.A and P.B.

2. The access mode of *x* in P.B is not *private*.

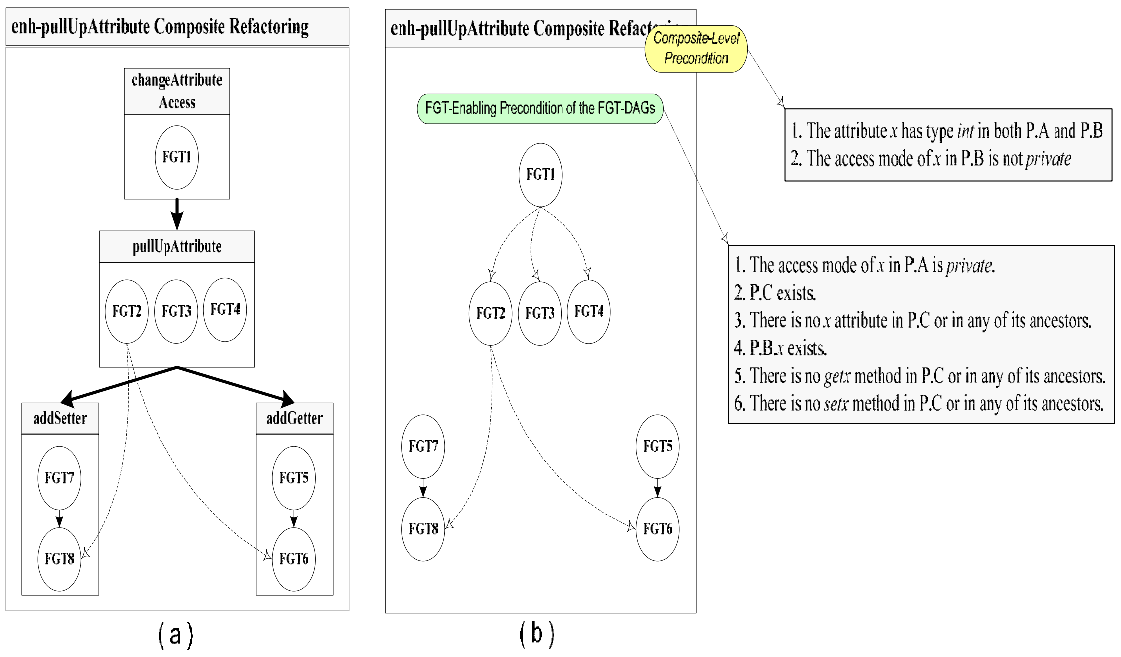
Since the **pullUpAttribute** is the only primitive refactoring in the composite which has refactoring-level preconditions, there are no other conjuncts in the composite-level precondition.

Since no reductions are possible when merging the various FGT-DAGs, Figure 10.8(b) shows the final result of the composite refactoring **enh-pullUpAttribute**. Note that in this case, it is represented as a single FGT-DAG. The composite-level precondition conjuncts are represented in a yellow box at the top of the composite. Note that the sequential dependencies between refactorings in the composite that were discovered by referring to refactoring-level pre- and postconditions are represented by making a link between all the leaf FGTs in the first primitive and all the root FGTs in the second primitive. In this example, the two primitive refactorings **changeOAMode** and **pullUpAttribute** have to be linked in this way. As a result, links are created between FGT1 and each of FGT2, FGT3, and FGT4. This ensures that at the refactoring time, none of the **pullUpAttribute's** FGTs will be executed until FGT1 is executed.

The FGT-enabling precondition conjuncts of the FGT-DAG are represented in a green box at the top of the composite. They can be inferred from its constituent FGTs as consisting of the following conjuncts:

1. The access mode of *x* attribute in P.A is *private*.
2. P.C exists.
3. There is no *x* attribute in P.C or in any of its ancestors.
4. P.B.*x* exists.
5. There is no *getx* method in P.C or in any of its ancestors.
6. There is no *setx* method in P.C or in any of its ancestors.

Checking the composite-level precondition as well as the FGT-enabling precondition of the FGT-DAG against the system in Figure 10.7(a) verifies that all the various conjuncts hold. Therefore the FGTs in Figure 10.8(b) may be applied to the system without the danger of rollback. The result is then the system depicted in Figure 10.7(b).


enh-pullUpAttribute Composite Refactoring
changeAttribute Access
FGT1
pullUpAttribute
FGT2
FGT3
FGT4
addSetter
FGT7
FGT8
addGetter
FGT5
FGT6
(a)
enh-pullUpAttribute Composite Refactoring
Composite-Level Precondition
FGT-Enabling Precondition of the FGT-DAGs
FGT1
FGT2
FGT3
FGT4
FGT7
FGT8
FGT5
FGT6
(b)

1. The attribute *x* has type *int* in both P.A and P.B
2. The access mode of *x* in P.B is not *private*

1. The access mode of *x* in P.A is *private*.
2. P.C exists.
3. There is no *x* attribute in P.C or in any of its ancestors.
4. P.B.*x* exists.
5. There is no *getx* method in P.C or in any of its ancestors.
6. There is no *setx* method in P.C or in any of its ancestors.

Figure 10.8: enh-pullUpAttribute composite refactoring

10.4 Reflection on this Chapter

This chapter has shown how composite refactorings can be built in the context of the FGT paradigm. It has been seen that FGT-DAGs can be merged, and that a composite-level precondition as well as FGT-enabling preconditions which are then used to avoid rollback. Furthermore, redundancies that may arise because of the merging of FGT-DAGs can be eliminated.

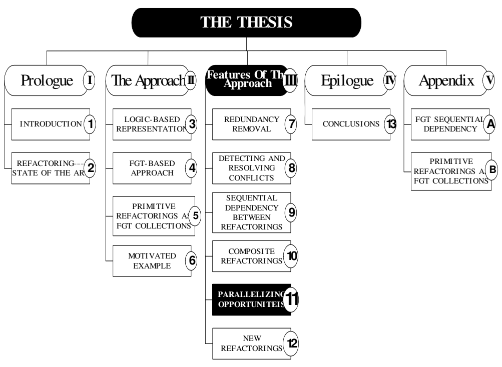
While the discussion has been in terms of an initial set of primitive refactorings, there is nothing to prevent the ideas developed in this chapter being carried over to compose composite refactorings from other composite refactorings. The approach developed in the previous chapter can then be used to determine sequential dependencies between such composites. As the size of composites grows, it may reasonably be conjectured that the scope for conflicts arising between them, and the scope for detecting reductions will increase. Again, these matters can be dealt with as described in chapters 7, 8 and 9.

Whether or not there will be a need for ever-larger and more complicated refactorings in the future, is a matter of conjecture. To the extent that there is, it would seem that the features described above will be of practical importance. What is clear, however, is that FGT-DAGs expose opportunities for parallel implementation. This will be discussed in the next chapter. The chapter after that will examine the implications of all of the above on providing end-user support for building new refactorings.

Chapter 11

PARALLELIZING OPPORTUNITIES

11.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITIES 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

In the FGT approach, opportunities for parallelizing are manifested at the time of refactoring and also during the process of reduction, detecting conflicts, determining sequential dependencies, and generating composites between refactorings. This is because of the ability of the approach to represent refactoring as a collection of FGTs, which are distributed among different FGT-DAGs according to their sequential dependency relations. These FGT-DAGs are independent and can be managed concurrently.

In previous approaches parallelizing are discussed at the level of refactorings. Given a chain of refactorings, Roberts in [70] pointed out that the dependency relationships between refactorings can be used to determine which sets of refactorings within the chain can be performed in parallel, and which ones must be performed sequentially. Each chain can be assigned to a separate processor.

Parallelizing in the proposed approach goes one level down by expressing parallelizing at the FGTs level, which offers the possibility of parallelizing the transformation inside one refactoring. The benefit of this can be easily seen, especially in respect of large refactorings such as composite refactorings with many FGTs inside it. For example, Figure 10.6(b) showed that the **encasulateAttribute** refactoring for the *Mark* attribute in the *College* system ends up with two FGT-DAGs, which can be applied and processed in parallel.

While it is beyond of the scope of this thesis to define parallel versions of the different algorithms developed throughout the thesis, the next section suggests, in overview, some of the ways in which parallelization can be exploited in the various FGT-related algorithms.

11.2 Parallelizing Opportunities

Parallelizing opportunities can be achieved in more than one place:

A. In **reduction algorithm**: As described in section 7.5 that the reduction algorithm works separately on each FGT-DAG. In a parallel version of the reduction algorithm, each FGT-DAG can be assigned to a separate processor.

B. In **conflict detection algorithm**: As described in section 8.4, the detection algorithm checks if there is a conflict between a given FGT in each FGT-DAG of refactoring X and the FGTs in all the other FGT-DAGs of refactoring Y. If a conflict is detected, then the algorithm has to resolve the conflict, either by withdrawing a refactoring, or by modifying an FGT-DAG in a refactoring. Various parallel versions of this algorithm can be developed. One is to assign separate processors to the FGT-DAGs in refactoring Y. Then each processor will run the detection algorithm described in section 8.4 to search conflicts between FGTs in its FGT-DAG and FGTs in FGT-DAGs of the other refactorings. Note that this parallel version assumes that any detected conflict will be resolved in the FGT-DAG of refactoring Y.

C. In **sequential dependency algorithm**: As described in section 9.4, the sequential dependency algorithm takes each FGT from refactoring X and checks if it has a sequential dependency relation with at least one FGT in refactoring Y. Here parallelizing can be done at the level of each FGT. One of the most fine-grained parallel versions of this algorithm would be to have one processor per FGT-pair to be tested. A processor first checks if there is a match between its pair of FGTs and one of the **uniDirSD** facts. If so, it terminates and declares a sequential dependency. Otherwise it continues to check for a match with **biDirSD** facts, determines the direction if one is found, declares a sequential dependency and terminates. A centralised scheduler should receive results and direct all running processes to abort when the first sequential dependency is reported.

D. At **refactoring time**: As explained in section 4.4.4, applying refactoring (primitive or composite) on the system can be done in two phases:

- In the first phase, the tool checks refactoring's precondition conjuncts against the system. To do that it checks first the refactoring-level precondition conjuncts, and if they are satisfied it checks the FGT-enabling preconditions of the various FGT-DAGs in the refactoring. Checking the FGT-enabling preconditions can be executed in parallel by assigning a processor to each one of the various FGT-DAGs of that refactoring. Each processor then will be responsible for checking the FGT-enabling precondition conjuncts of its FGT-DAG. If all the precondition conjuncts are satisfied in all processors then the tool goes to the second phase.

- In the second phase, the tool applies the FGTs of the refactoring under consideration. This stage also can be done in parallel by letting each processor apply the FGTs in its FGT-DAG to the system.

The foregoing describes in overview the potential for parallelizing at the FGT-DAG level. However, it should be noted that more fine-grained parallelization would also be possible within an FGT-DAG. In this case, FGTs on separate branches of the FGT-DAG could run in parallel pipelines with one another, but would then have to synchronize appropriately on FGT nodes at which there is more than one inbound arc.

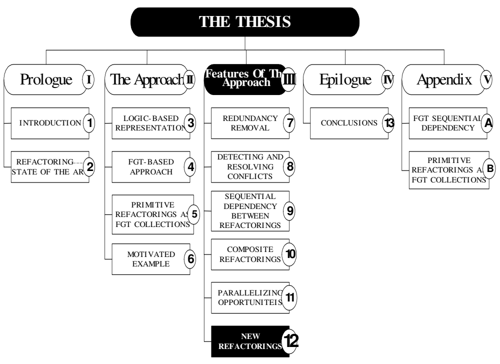
11.3 Reflection on Parallelization

Although some may question the relevance of parallelizing the refactoring task and associated algorithms in the contemporary world of refactoring, this would seem to be a rather short-sighted view. On the one hand, the matter of potential for parallelizing computational tasks has always been of theoretical interest in computer science. On the other hand, it is now widely acknowledged that current trends in chip design are in the direction of increasing the number of cores per chip. Indeed, in recent years, Intel and others have emphasized the importance of adapting computer science curricula to prepare students for a future in which parallel/concurrent programming will become ever-more dominant. This chapter has suggested that an FGT-based approach to refactoring seems well-adapted to such a future.

Chapter 12

NEW REFACTORINGS

12.1 Introduction


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

Kniesel and Koch [38] point to a dilema that confronts the developer of a refactoring tool. On the one hand, user needs are not limited to a core of custom refactorings that can be embedded into a tool. In fact, the type and complexity of refactorings needed varies according to areas of application and needs evolve over time. In this regard, they mention applications like "refactoring to patterns" [8, 37] and "refactoring to aspects" [30]. On the other hand, they note that tools lack user-definable refactorings. They point out that:

> *"[This] lack of user-definable refactorings is equally unsatisfactory for tool providers and for their users. For tool providers, because they must continuously invest time and money in the never-ending evolution of refactorings. For users, because they are forced either to wait for some future release, hoping that it will provide the missing functionality, or to implement their own custom refactorings. However, the latter is not a real option for most users. After all, most developers are interested in refactoring as a means of speeding their own development activities, not as an additional development task within an anyway much too tight schedule."*

One of the solutions for the above problem is for a refactoring tool to provide the end user with a facility for composing larger refactorings from primitive ones. This possibility was described in detail in chapter 10. Unfortunately, the provision of such a facility in a tool is not be sufficient. Providing a set of primitive refactorings that can be executed in sequence in order to achieve a complex effect, is not the same as providing users with the ability to define their own refactorings. Some cases may exist where the end user needs to build a new refactoring that cannot be constructed by just using the primitive refactorings that have been implemented in the refactoring tool.

12.2 Example

Return to the LAN motivated example presented in chapter 6. As before, suppose that one of the proposed enhancements to make to the class diagram shown in Figure 12.1, is to pull up the *accept* method from the subclasses *FileServer* and *PrintServer* to their superclass *Server*. The motivation for this refactoring is that the *accept* method in the two subclasses are identical, and it is preferred to pull it up to the common superclass for the reasons described in section 5.3.2.6.

Assume that, in contrast to chapter 6, the *accept* method accesses the *public* method, *process*, in the two subclasses. If the *accept* method is moved from the subclasses to the superclass, this access (to the *process* method) will not be visible from the superclass. In fact, at the code-level such a move would result in a "*process method is undefined*" compiler error[2].

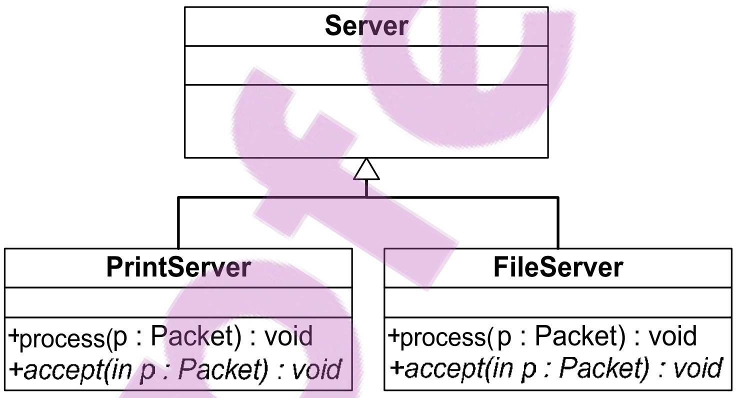

Server
PrintServer
+process(p : Packet) : void
+accept(in p : Packet) : void
FileServer
+process(p : Packet) : void
+accept(in p : Packet) : void

Figure 12.1: Part of the LAN system's class diagram

To avoid such problems, one of the precondition conjuncts of the refactoring **pullUpMethod** requires that all the references made by the pulled up method to the other object elements in the system must be visible from the superclass. According to this precondition, pulling up the method *accept* in the example will be rejected.

Suppose that the developer considers that this precondition is very restrictive. There are ways in which such a refactoring can be applied without affecting the behaviour of the system. With reference to the *accept* method in the present case, one solution is to define an empty method with name *process* in the superclass, which has the same signature as the *process* method in

[2] Note that the fact that the *process* method is *public* is incidental to the argument here. The argument would be the same if its access mode had been *protected* or *private*.

the subclasses. The result will be that the referenced made by the *accept* method in the superclass to the *process* method will be valid now.

Therefore, the idea is to define in the superclass all the methods in the subclasses that the pulled up method references in the subclasses. (It is recognised that a cleaner solution would be to define an abstract method in the superclass, but these have not been considered in the present work.)

Defining these methods in the superclass will not affect the behaviour of the system because the newly defined methods are not referenced by any other object elements. Also, they are empty and will be overridden by the original members defined in the subclasses.

Note that the above enhancement to the refactoring **pullUpMethod** is valid only if the access mode of the *process* method in the subclasses is *public* or *protected*—i.e. it may not be *private*.

Suppose that the end user wants to create a new refactoring that takes into consideration the above enhancement to the **pullUpMethod** refactoring. The new refactoring should work as follows:

a. The set of methods in the subclasses referenced by at least one subclass version of the method to be pulled up should be noted. Each element of this set should have the same signature in the subclasses in which it occurs. No element of this set should have a *private* access mode in any of the subclasses. For each of these methods, an empty method with the same signature as defined in the subclasses should be added into the superclass. The access mode of each method should not be more general than the access modes of the corresponding versions of the method in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

b. The set of attributes in the subclasses referenced by at least one subclass version of the method to be pulled up should be noted. No element of this set should have a *private* access mode in any of the subclasses. The access mode of each attribute should not be more general than the access modes of the corresponding versions of the attribute in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

It is clear that it is impossible to accomplish this by trying to compose a sequence (collection) of the primitive refactorings presented in Table 4.1, which means that the approach of building a composite refactoring presented in chapter 10 will not work here. A new approach is required.

12.3 New Refactorings in the FGT-Based Approach

A refactoring tool based on FGTs can, in principle, enabled users to build their own refactorings without needing to write code. Instead, they would rely on the set of the low-level FGTs proposed in the thesis, as well as various algorithms discussed in earlier chapters. In outline, what is needed is a small domain specific language (DSL) whose semantics allows for (a) selection from the tool's set of FGTs; (b) simple conditional and looping statements; (c) specification of a new refactoring's refactoring-level preconditions and (d) simple storage and retrieval of named and parameterised procedures. The body of the procedure would then consist of a sequence of FGTs, some of which may need to be conditionally included, depending on the system eventually to be refactored. The new refactoring can be saved as a named procedure with a list of input parameters. "Compilation" of the procedure would involve instantiating the refactoring into an FGT-list for a given system, decomposing the list into a set of FGT-DAGs, reducing FGTs where required, identifying and possibly resolving conflicts, and computing the refactorings FGT-enabling precondition for that system. Although it is beyond the scope of this thesis to develop such a DSL, its successful implementation would go a long way to resolving the refactoring dilemma referred to by Kniesel and Koch and cited above.

In the absence of such a DSL, the in which a new FGT-based can be implemented will now be illustrated. The refactoring

enh-pullUpMethod(*SubClassesNames, Methn, MethTList*)

takes into consideration the enhancement proposed in section 12.2, and consists of the following sequence of FGTs:

1. addObject(*SupPn, SupCn, Methn, _, _, MethRType, OAMode, MethTList, method*)

2. **For** each relational element between *Methn* as a source and *Methx* as destination where the access mode of *Methx* is *public* or *protected* and *Methx* is defined in the same class as *Methn* **do** {
   addObject(*SupPn, SupCn, Methx, _, _, MethxRType, MOAMode, MethxTList, method*) }

3. **For** each relational element between *Methn* as a source and *Attx* as destination where the access mode of *Attx* is *public* or *protected* and *Attx* is defined in the same class of *Methn* **do** { addObject(*SupPn, SupCn, Attx, _, _AttxDType, AAMode, _attribute*) }

4. **For** each subclass in the *SubClassesNames* list **do** {

   deleteObject(*SubPn*$_i$,*SubCn*$_i$, *Methn,_, MethTList, method*) }

The differences between the new refactoring **enh-pullUpMethod** and the refactoring **pullUpMethod** presented in section 5.3.2.6 are in steps 2 and 3. These are not found in the **pullUpMethod**. In step 2, the method members added to superclass are all those methods defined in the subclasses *SubClassesNames*, referenced by the *Methn*, and their access mode is *public* or *protected*. Similar, checks are done in step 3 with respect to attribute members. Note that the value of the arguments *MOAMode* in step 2 and *AAMode* in step 3 are calculated according to the rule described in section 12.2.

After applying the new refactoring

enh-pullUpMethod(*['FileServer', 'PrintServer'], accept, ['Packet']*)

to the class diagram shown in Figure 12.1, the class diagram will be restructured as shown in Figure 12.2.

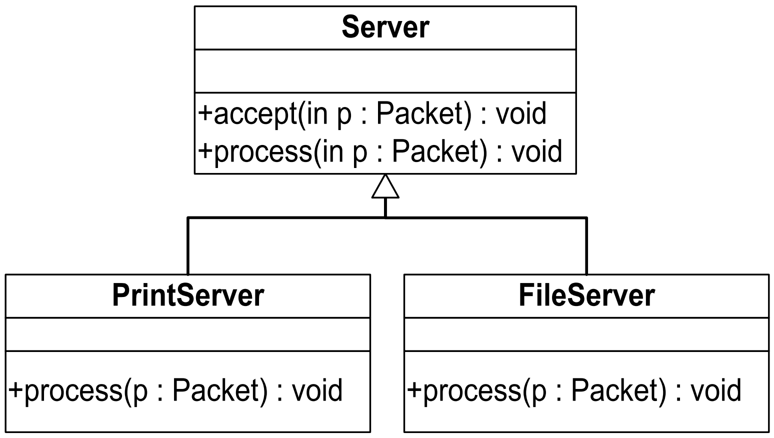

Server
+accept(in p : Packet) : void
+process(in p : Packet) : void
PrintServer
+process(p : Packet) : void
FileServer
+process(p : Packet) : void

Figure 12.2: Part of the LAN system's class diagram after enh-pullUpMethod

12.4 Reflection on this Chapter

Giving end users the ability to use FGTs to construct their own refactorings has the following advantages:

1. The user is not restricted to use the list of primitive refactorings implemented in the tool for building other refactorings. Instead, a much wider variety of refactorings can be built, due to the more comprehensive semantics of FGTs.

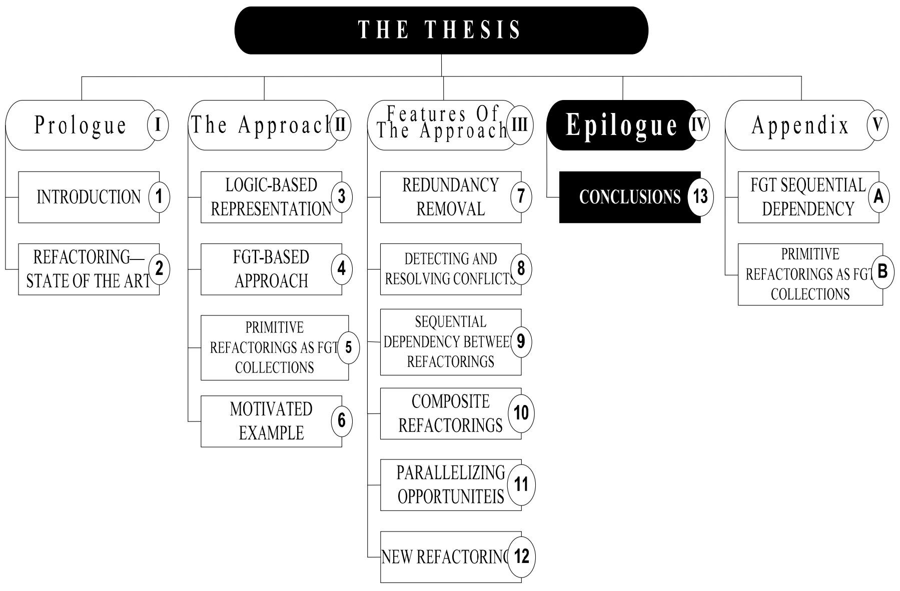
2. Because pre- and postconditions of FGTs can be stored in the refactoring tool, the user is absolved from articulating them again when creating new refactorings. In fact, after the desired sequence of FGTs has been found, FGT-enabling preconditions for the various FGT-DAGs of the new refactoring can be automatically computed. Note, however, that articulating refactoring-level preconditions of the refactoring remains the responsibility of the user.

3. As mentioned above, there is no need from the user to write a pure code.

4. Because the new refactoring will be built as a collection of FGTs, all features presented in the thesis for such representation can be applied to the new refactorings.

Part IV

Epilogue

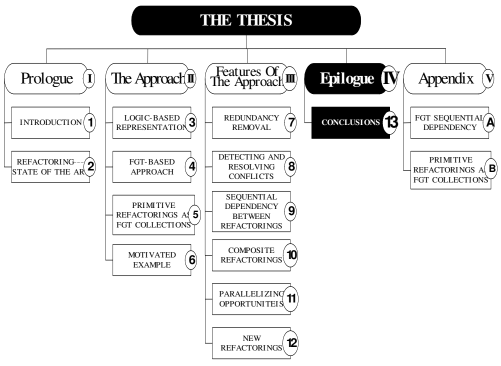

THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING—STATE OF THE ART 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORING 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

Chapter 13

CONCLUSIONS

13.1 Summary

This work can be summarized as the follows:


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING--- STATE OF THE AR 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS A FGT COLLECTIONS B

A. In part I—which includes chapter one and two—an introduction to refactoring, problems associated with it, and proposed solutions are discussed. A survey of previous work in refactoring topics related to the thesis was presented. The role of evolution in the system life cycle, the levels of the system artifacts where the refactorings can be applied, and the different refactorings formalism techniques were covered in the survey.

B. In part II—which includes chapters three to six—a new formalism to represent refactorings at the design level is presented. The new formalism defines and executes model refactorings as a set of FGTs ordered in one or more FGT-DAGs. It also introduces refactoring pre- and postcondition conjuncts at two different levels (FGT-level and refactoring-level). Detailed descriptions of the set of FGTs that are used in the approach together with their set of preconditions are also presented. A logic-based representation was presented in this part of UML class diagrams, of related objects, and of reference information extracted from the code-level of the system under consideration. The part also discussed the relationship between the proposed FGT paradigm and primitive- as well as composite refactorings. It was shown that FGTs can be the core of a refactoring system in which a wide range of refactorings can be constructed and represented by a collection of these FGTs. To show the feasibility of the approach and its ability to represent refactorings, FGT representations of twenty-nine common primitive refactorings were presented in chapter 5. The chapter also discussed the set of precondition conjuncts of each refactoring and how these precondition conjuncts are related to the precondition conjuncts of their associated FGTs. At the end of this part, in chapter 6, a motivated example was given. The example, "*A simulation of a Local Area Network (LAN)*", is frequently used for teaching refactoring. The chapter shows how the UML class diagram of the LAN system (with the additional reference information) is represented as logic-terms. In addition to the twenty-nine primitive refactorings presented in chapter 5, chapter 6 shows how two other well-known composite refactorings (**encapsulateAttribute** and **createClass**) are represented in the proposed formalism.

C. In part III—which includes chapters seven to twelve—various features of the proposed formalism were explored. Chapter 7 showed how redundancy between FGTs in the same FGT-DAG can be removed. For that a **reduction** algorithm was developed. This feature reduces the number of FGTs and the associated number of refactoring precondition conjuncts, thus increasing the efficiency of refactoring. In addition, the number of sequential dependencies between the different FGTs inside the refactoring will be reduced and the pseudo-conflicts will be eliminated. Chapter 8 showed how conflict freedom can be established using the **detectResolveConflict** algorithm that was developed. Three different kinds of conflicts between pairs of refactorings were described and treated: ordering-conflicts (where conflict can be resolved by ordering one of the refactorings before the other); cancelling-conflicts (where conflict can only be resolved by withdrawing one of the refactorings); and removable-conflicts (where conflicts can be resolved by appropriately modifying FGTs that comprise one of the refactorings). Then, in chapter 9, finding the sequential dependency between two refactorings was discussed. For that a **sequentialDependency** algorithm was developed. Also in this chapter, the deadlock and the ambiguity terms were introduced and treated properly. An FGT-based approach to deal with composite refactorings was introduced in chapter 10. The scope for parallelizing FGT-based refactoring at various levels was discussed in chapter 11. Parallelizing suggestions (extensions) for the different algorithms presented in the thesis were explored in overview. Finally, in chapter 12, the feature of giving the end users the ability to create their own FGT-based refactorings without having to write a code is presented. The proposal of developing a DSL that is based on the FGT paradigm was made, and an illustration was provided of how FGTs can be used to build a refactoring that does not consist of a number of primitive refactorings.

Over the past few years, various aspects of this work have been published in peer-reviewed conferences and workshops:

1. [73] represents the initial work that led to the investigation of the new refactoring formalism using the FGT paradigm. The FGT methodology was briefly introduced in the paper.

7. [74] presents the algorithm to find automatically the optimal ordering in which to apply a batch of refactorings. The proposed algorithm detects implicit sequential dependencies, resolves conflicts between the different refactorings in the batch and minimizes the number of refactoring operations by removing the redundant ones. The algorithm is based on the FGT paradigm described thoroughly in this work.

2. [75] extends the work done in [73] by introducing a new formal definition of refactorings that supposed to work at the UML class diagrams. Feasibility and features of the new approach are explored in the paper.

In the next section, we provide a conclusion to our work.

13.2 Conclusions

The followings has been achieved in this thesis:

1. **FGT-Based Refactoring Formalism Technique:** This work has established a new technique to formalize refactorings applied at the design level (*UML class diagrams in specific*). The new formalism is based on the so-called FGT paradigm. The feasibility and features of the new approach are discussed thoroughly in the thesis. A detailed set of FGTs together with their set of precondition conjuncts were defined in the work. These FGTs are at the core of the refactoring formalism. Based on the new formalism, a design level refactoring can be seen as:

   "A collection of FGTs ordered in one or more FGT-DAGs with a set of pre- and postcondition conjuncts installed at the level of the whole refactoring and a set of pre- and postcondition conjuncts installed at the level of each FGT"

   Several common refactorings already available in the literature (twenty-nine primitive refactorings and two composite refactorings) have been presented in terms of such FGT-DAGs.

2. **Logic-Based UML Class Diagrams Representation:** The work has shown how UML class models can be represented as a set of logic-terms (*facts in Prolog*). The proposed representation can be used for refactoring (as done in this thesis). However, it can also be used as a basis for issuing Prolog queries about a UML system.

3. **Remove Redundancy:** The work has defined a method for removing redundancy at the FGT-level in refactorings that may grow complex as they composed into ever-more larger ones over time.

4. **Detect and Resolve Conflict:** Additionally, the work has defined a method for detecting conflicts between refactorings. The fact that the detection is at the more fine-grained FGT-level as opposed to refactoring-level, means that the source of conflicts can be accurately pin-pointed and resolved by manipulating FGTs rather than refactorings.

5. **Find Sequential Dependency:** A method for finding the sequential dependency that may occur between refactorings has defined in the work. To do that, the method is also based on the idea of finding the sequential dependency at the level of FGTs.

6. The concept of a **"refactoring deadlock"** has analysed, and a method to detect a deadlock between two refactorings has been proposed.

7. Conditions under which **"ambiguity"** in the sequential dependency between two refactorings arises, has been identified and catalogued. A method to solve such ambiguity has been proposed.

8. **Composite Refactorings:** The work has introduced a methodology to deal with composite refactorings in an FGT context. The methodology constructs the composite refactoring from a collection of FGTs with a set of composite-level pre- and postcondition conjuncts. Because the resulting composite is expressed in terms of FGTs, the composite can be analysed with respect to conflict, redundancy, sequential dependency and parallelizing opportunities—just as any other FGT-based refactoring. Furthermore, by suitably checking preconditions against an existing system, rollback can be avoided—just as in the case of previous approaches.

8. **Parallelizing Opportunity:** The work naturally exposes parallelizing opportunities at the time of refactoring or during the process of detecting conflicts, removing redundancies and finding sequential dependencies between refactorings. This is basically because the FGTs for a refactoring are classified into FGT-DAGs, depending on the sequential dependency between these FGTs. These FGT-DAGs are independent and can be managed concurrently.

9. **New Refactorings:** The work has established the basic foundation for giving the end users the ability to create new refactorings whose semantics is constrained, not by the selection of existing refactorings that have been implemented in the tool, but rather by the semantics of

the FGTs that have been predefined in the tool. This can be based on a DSL that can be used to create a more complex refactorings.

The differences between refactoring based on an FGT paradigm and those of alternative approaches are summarized in Table 13.1.

Table 13.1: A comparison between FGTs-based and alternative formalisms

Alternative Formalisms	FGTs-Based Formalism
→ Refactoring is a black box.	→ Refactoring is a collection of FGTs ordered in one or more FGT-DAGs.
→ Refactoring precondition conjuncts are defined at one level. (The same for postcondition conjuncts)	→ Refactoring precondition conjuncts are defined at two different levels. (The same for postcondition conjuncts)
→ No possibility of knowing which part of refactoring causes the conflicts. Therefore, it is difficult to resolve these conflicts.	→ Conflicts are detected at the level of FGTs. These conflicts can be resolved.
→ Less parallelizing opportunities.	→ More parallelizing opportunities.
→ Difficult for end users to build their own refactorings because there is a need to write a code.	→ Building new refactorings can be done by using the list of the proposed FGTs without a need to write a code.
→ Redundancy can only be removed at a refactoring-level.	→ Redundancy between FGTs can be removed.
→ No possibility to know at what specific point or points two refactorings are sequentially dependent.	→ Ability to know at what point or points two refactorings are sequentially dependent.

In general, there will be more FGTs than there are refactorings, and therefore more computational operations. However, this additional computational cost buys more flexibility—including, and especially, the flexibility afforded to the end user to define a wider range of refactorings than is possible when relying on primitive refactorings as building blocks. In the contemporary world of high-speed processors, and the relatively small scale of entities to be processed in a design level refactoring applications (typically in the order of thousands rather than millions or billions) the additional processing cost does not seem to be a significant factor. Moreover, it should also be borne in mind that the additional computational cost can be offset against the enhanced scope for parallelizing operations afforded by the FGT paradigm, where one can rely on the ever-increasing number of multi-core processors available on contemporary chips.

The next section considers further directions that this work could take.

13.3 Future Work

There are a variety of future challenges that require further investigation. Each subsection below contains a list of projects that could be undertaken by future researchers.

1. UML Meta-Model Extension: The work in the thesis is based on the simplified UML meta-model shown in Figure 1.5. For a full, mature and ready-to-use refactoring tool, an extension of the meta-model is needed to deal with constructs such as interfaces, aggregations, constructors and so on. It is to be expected that new dependencies and conflicts between the different FGTs will be introduced, and ways will have to be found deal with these.

2. Different Types of Software Artifacts: The discussion of the proposed approach in our work was based on applying refactorings to UML class diagrams. It may be possible to extend the approach to a wider range of UML modeling notations such as state and sequence diagrams. It may also be possible to extend the approach to the code-level, to database schemas, to software architectures or to the software requirements' levels. A more thorough investigation into these possibilities is needed, both in terms of feasibility and in terms of desirability.

3. Consistency: Throughout this work, the refactorings are reflected at the UML class diagram level. Ideally, the modifications should also be reflected on the other UML models affected by the refactoring, as well as on the code-level implementation of the system. This is because it is important to keep the different system models and code consistent with one another. Clearly, further research in this direction would be beneficial.

4. Removable-Conflicts: In chapter 8, three different kinds of conflicts between pairs of refactorings were described and treated: ordering-conflicts; cancelling-conflicts; and removable-conflicts. The resolving procedure for the first two kinds of conflicts is straightforward as discussed in chapter 8. Resolving the third kind of conflicts (removable-conflicts) need more attention. It is feasible to identify FGT pairs in the two refactorings that constitute removable-conflicts and it is also possible to offer guidelines about how one of the FGTs in the pair may be changed. However, there is no guarantee that resolving a removable-conflict will not introduce other conflicts. More investigation is needed on how to deal with detected removable-conflicts.

5. Deadlock Algorithm Extension: The deadlock algorithm presented in section 9.5 is concerned with finding a deadlock between two refactorings. However, it does not deal with the fact that a deadlock may arise from circular sequential dependency relationships. For example, the following scenario leads to a deadlock:

$$A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D,\ D \rightarrow A$$

A proper extension to the proposed deadlock algorithm should be investigated to deal with such cases, taking into consideration the algorithmic efficiency in the proposed solution.

6. Parallelizing Algorithms: Chapter 11 discussed the different parallelizing opportunities the new approach can open. Actual implementation of parallel algorithms should be investigated.

7. Larger scale example: The scope for parallelization has been explained in chapter 11. However, in order to explore the potential benefits of parallelization, large scale real-life examples should be investigated.

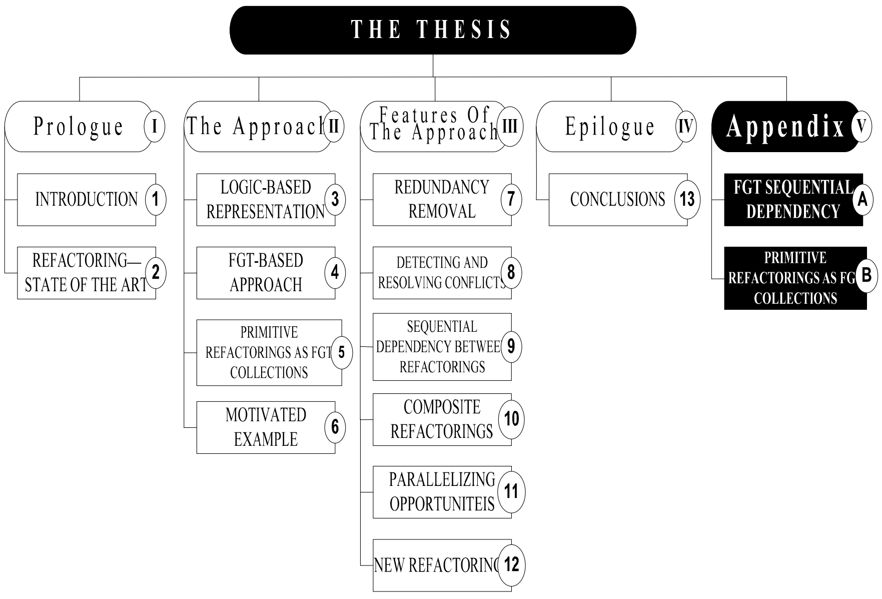
8. Domain Specific Language (DSL): Research is needed into developing a fully-fledged DSL for end users to create their own refactorings. The proposed language will have the features such as the following:

- A set of fully implemented FGTs with their pre- and postcondition conjuncts. This set will be ready for the user to select and use to construct a refactoring.
- A set of language constructs like (for example, if-statements, for-loops, etc). The syntax of these constructs should be specifically tailored to accommodate the FGT paradigm, they should be intuitive, easy to use, and should be sufficiently expressive for the user to assemble the desired sequence of FGTs to represent the intended refactoring.
- Support structures for the user easily to formulate refactoring-level pre- and postconditions. In the Prolog prototype, these took the form of procedures such as existsObject(--), supclass(--), subclass(--), isReferenced(--), etc.

While the development of such a DSL together with an environment in which it can be used is a non-trivial task, it seems like worthwhile endeavour that will maximally uncover the benefits to be derived from refactoring based on the FGT paradigm.

Part V

Appendix

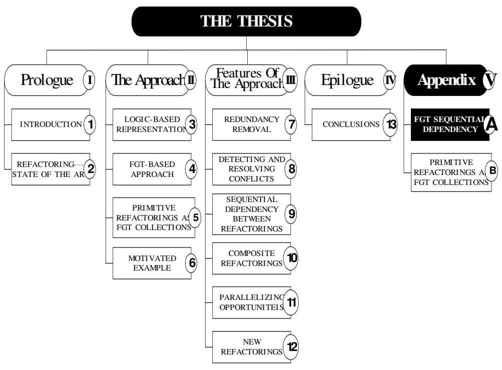

THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING— STATE OF THE ART 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORING 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

Appendix A

FGT SEQUENTIAL DEPENDENCY

A.1 Uni-Directional Sequential Dependencies

In the table below, all the uni-directional sequential dependencies between the different FGTs proposed in our approach are catalogued. The information in the table is a continuation of the discussion in section 4.3.2. Each row in the table represents the following sequential dependency: $FGT_x \rightarrow FGT_y$, where FGT_y is sequentially dependent on FGT_x. Note that each numbered row in the table corresponds to a numbered arc in Figure 4.1 that represents a uni-directional sequential dependency.


THE THESIS
Prologue I
INTRODUCTION 1
REFACTORING STATE OF THE AR 2
The Approach II
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
Features Of The Approach III
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
Epilogue IV
CONCLUSIONS 13
Appendix V
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

1.	changeODefType(P,C,M,PR,PLT,ObjT,_,ONewDT) → changeODefType(P,C,M,PR,PLT, ObjT,ONewDT,_)
2.	changeOAMode(P,C,M,PR,PLT,ObjT,_,ONewAM) → changeOAMode(P,C,M,PR,PLT,ObjT,ONewAM,_)
3.	deleteRelation(_,P,C,M,PR,PLT,Ftype,_,_,_,_,_,_,_) → changeODefType(P,C,M,PR,PLT,Ftype,_,_)
4.	deleteRelation(_,_,_,_,_,_,_,P,C,M,PR,PLT,Totype,_) → changeODefType(P,C,M,PR,PLT,Totype ,_,_)
5.	deleteRelation(_,P,C,M,PR,PLT,Ftype,_,_,_,_,_,_,_) → changeOAMode(P,C,M,PR,PLT,Ftype,_,_)
6.	deleteRelation(_,_,_,_,_,_,_,P,C,M,PR,PLT,Totype,_) → changeOAMode(P,C,M,PR,PLT,Totype ,_,_)
7.	deleteRelation(_,P,C,M,PR,PLT,Ftype,_,_,_,_,_,_) → deleteObject(P,C,M,PR,PLT,Ftype)
8.	deleteRelation(_,_,_,_,_,_,_,P,C,M,PR,PLT,Totype,_) → deleteObject(P,C,M,PR,PLT,Totype)
9.	changeODefType(P,C,M,PR,PLT,ObjT,_,_) → addRelation(_,P,C,M,PR,PLT,ObjT,_,_,_,_,_,_,_)
10.	changeODefType(P,C,M,PR,PLT,ObjT,_,_) → addRelation(_,_,_,_,_,_,_,P,C,M,PR,PLT,ObjT ,_)
11.	changeOAMode(P,C,M,PR,PLT,ObjT,_,_) → addRelation(_,P,C,M,PR,PLT,ObjT,_,_,_,_,_,_,_)
12.	changeOAMode(P,C,M,PR,PLT,ObjT,_,_) → addRelation(_,_,_,_,_,_,_,P,C,M,PR,PLT,ObjT,_)
13.	renameObject(P,C,M,PR,PLT,parameter,X) → addRelation(_,P,C,M,X,PLT,parameter,_,_,_,_,_,_,_)
14.	renameObject(P,C,M,_,_,attribute,X) → addRelation(_,P,C,X,_,_,attribute,_,_,_,_,_,_,_)
15.	renameObject(P,C,M,_,PLT,method,X) → addRelation(_,P,C,X,_,PLT,method,_,_,_,_,_,_,_)
16.	renameObject(P,C,_,_,_,class,X) → addRelation(_,P,X,_,_,_,class,_,_,_,_,_,_,_)
17.	renameObject(P,C,M,PR,PLT,parameter,X) → addRelation(_,_,_,_,_,_,_,P,C,M,X,PLT,parameter,_)
18.	renameObject(P,C,M,_,_,attribute,X) → addRelation(_,_,_,_,_,_,_,P,C,X,_,_,attribute,_)
19.	renameObject(P,C,M,_,PLT,method,X) → addRelation(_,_,_,_,_,_,_,P,C,X,_,PLT,method,_)
20.	renameObject(P,C,_,_,_,class,X) → addRelation(_,_,_,_,_,_,_,P,X,_,_,_,class,_)
21.	renameObject(P,C,M,PR,PLT,parameter,PR1) → deleteRelation(_,P,C,M,PR1,PLT,parameter,_,_,_,_,_,_,_)
22.	renameObject(P,C,M,_,_,attribute,PR1) → deleteRelation(_,P,C,PR1,_,_,attribute,_,_,_,_,_,_,_)
23.	renameObject(P,C,M,_,PLT,method,PR1) → deleteRelation(_,P,C,PR1,_,PLT,method,_,_,_,_,_,_,_)
24.	renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,P,PR1,_,_,_,class,_,_,_,_,_,_,_)

25. renameObject(P,C,M,PR,PLT,parameter,PR1) → deleteRelation(_,_,_,_,_,_,_,P,C,M,PR1,PLT,parameter,_)
26. renameObject(P,C,M,PR,_,attribute,PR1) → deleteRelation(_,_,_,_,_,_,_,P,C,PR1,_,_,attribute,_)
27. renameObject(P,C,M,PR,PLT,method,PR1) → deleteRelation(_,_,_,_,_,_,_,P,C,PR1,_,PLT,method,_)
28. renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,_,_,_,_,_,_,P,PR1,_,_,_,class,_)
29. renameObject(P,C,M,PR,PLT,parameter,X) → changeODefType(P,C,M,X,PLT,parameter,_,_)
30. renameObject(P,C,M,_,_,attribute,X) → changeODefType(P,C,X,_,_,attribute,_,_)
31. renameObject(P,C,M,_,PLT,method,X) → changeODefType(P,C,X,_,PLT,method,_,_)
32. renameObject(P,C,_,_,_,class,X) → changeODefType(P,X,_,_,_,class,_,_)
33. renameObject(P,C,M,PR,PLT,parameter,X) → changeOAMode(P,C,M,X,PLT,parameter,_,_)
34. renameObject(P,C,M,_,_,attribute,X) → changeOAMode(P,C,X,_,_,attribute,_,_)
35. renameObject(P,C,M,_,PLT,method,X) → changeOAMode(P,C,X,_,PLT,method,_,_)
36. renameObject(P,C,_,_,_,class,X) → changeOAMode(P,X,_,_,_,class,_,_)
37. renameObject(P,C,_,_,_,class,PR1) → addObject(P,PR1,M,_,_,_,_,_,attribute)
38. renameObject(P,C,_,_,_,class,PR1) → addObject(P,PR1,M,_,_,_,_,[],method)
39. renameObject(P,C,M,_,PLT,method,PR1) → addObject(P,C,PR1,PR,_,_,_,PLT,parameter)
40. renameObject(P,C,_,_,_,class,PR1) → changeOAMode(P,PR1,M,_,_,attribute,_,_)
41. renameObject(P,C,_,_,_,class,PR1) → changeOAMode(P,PR1,M,_,PLT,method,_,_)
42. renameObject(P,C,M,_,PLT,method,PR1) → changeOAMode(P,C,PR1,PR,PLT,parameter,_,_)
43. renameObject(P,C,_,_,_,class,PR1) → changeODefType(P,PR1,M,_,_,attribute,_,_)
44. renameObject(P,C,_,_,_,class,PR1) → changeODefType(P,PR1,M,_,PLT,method,_,_)
45. renameObject(P,C,M,_,PLT,method,PR1) → changeODefType(P,C,PR1,PR,PLT,parameter,_,_)
46. renameObject(P,C,_,_,_,class,PR1) → deleteObject(P,PR1,M,_,_,attribute)
47. renameObject(P,C,_,_,_,class,PR1) → deleteObject(P,PR1,M,_,PLT,method)
48. renameObject(P,C,M,_,PLT,method,PR1) → deleteObject(P,C,PR1,PR,PLT,parameter)
49. renameObject(P,C,_,_,_,class,PR1) → addRelation(_,P,PR1,M,_,_,attribute,_,_,_,_,_,_,_)
50. renameObject(P,C,_,_,_,class,PR1) → addRelation(_,_,_,_,_,_,P1,PR1,M,_,_,attribute,_)
51. renameObject(P,C,_,_,_,class,PR1) → addRelation(_,P,PR1,M,_,PLT,method,_,_,_,_,_,_,_)
52. renameObject(P,C,_,_,_,class,PR1) → addRelation(_,_,_,_,_,_,P1,PR1,M,_,PLT,method,_)
53. renameObject(P,C,M,_,PLT,method,PR1) → addRelation(_,P,C,PR1,PR,PLT,parameter,_,_,_,_,_,_,_)
54. renameObject(P,C,M,_,PLT,method,PR1) → addRelation(_,_,_,_,_,_,P,C,PR1,PR,PLT,parameter,_)
55. renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,P,PR1,M,_,_,attribute,_,_,_,_,_,_,_)
56. renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,_,_,_,_,_,P1,PR1,M,_,_,attribute,_)
57. renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,P,PR1,M,_,PLT,method,_,_,_,_,_,_,_)
58. renameObject(P,C,_,_,_,class,PR1) → deleteRelation(_,_,_,_,_,_,P1,PR1,M,_,PLT,method,_)
59. renameObject(P,C,M,_,PLT,method,PR1) → deleteRelation(_,P,C,PR1,PR,PLT,parameter,_,_,_,_,_,_,_)
60. renameObject(P,C,M,_,PLT,method,PR1) → deleteRelation(_,_,_,_,_,_,P,C,PR1,PR,PLT,parameter,_)
61. addObject(P,C,_,_,_,_,_,PLT,class) → addObject(P,C,M,_,_,_,_,PLT,attribute)
62. addObject(P,C,_,_,_,_,_,PLT,class) → addObject(P,C,M,_,_,_,_,PLT,method)
63. addObject(P,C,M,_,_,_,_,PLT,method) → addObject(P,C,M,PR,_,_,_,PLT,parameter)
64. addObject(P,C,M,PR,_,_,_,PLT,ObjT) → addRelation(_,P,C,M,PR,PLT,ObjT,_,_,_,_,_,_)
65. addObject(P1,C1,M1,PR1,_,_,_,PLT,ObjT) → addRelation(_,_,_,_,_,_,P1,C1,M1,PR1,PLT,ObjT,_)
66. addObject(P,C,M,PR,Oldtype,_,_,PLT,ObjT) → changeODefType(P,C,M,PR,PLT,ObjT,Oldtype,Newtype)
67. addObject(P,C,M,PR,_,_,Oldmode,PLT,ObjT) → changeOAMode(P,C,M,PR,PLT,ObjT,OldMd, NewMd)
68. deleteObject(P,C,M,PR,PLT,parameter) → deleteObject(P,C,M,_,PLT,method)
69. deleteObject(P,C,M,_,PLT,method) → deleteObject(P,C,_,_,PLT,class)
70. deleteObject(P,C,M,_,PLT,attribute) → deleteObject(P,C,_,_,PLT,class)
71. renameObject(P,C,M,PR,PLT,parameter,X) → renameRelation(_,P,C,M,X,PLT,parameter,_,_,_,_,_,_,_,_)
72. renameObject(P,C,M,_,_,attribute,X) → renameRelation (_,P,C,X,_,_,attribute,_,_,_,_,_,_,_,_)
73. renameObject(P,C,M,_,PLT,method,X) → renameRelation (_,P,C,X,_,PLT,method,_,_,_,_,_,_,_,_)
74. renameObject(P,C,_,_,_,class,X) → renameRelation (_,P,X,_,_,_,class,_,_,_,_,_,_,_,_)

75.	renameObject(P,C,M,PR,PLT,parameter,X) → renameRelation (_,_,_,_,_,_,_,P,C,M,X,PLT,parameter,_,_)
76.	renameObject(P,C,M,_,_,attribute,X) → renameRelation (_,_,_,_,_,_,P,C,X,_,_,attribute,_,_)
77.	renameObject(P,C,M,_,PLT,method,X) → renameRelation (_,_,_,_,_,_,_,P,C,X,_,PLT,method,_,_)
78.	renameObject(P,C,_,_,_,class,X) → renameRelation (_,_,_,_,_,_,_,_,P,X,_,_,_,class,_,_)
79.	* deleteObject(P1,C1,M,_,_,attribute) → addObject(P2,C2,M,_,_,_,_,_,attribute)
80.	* deleteObject(P1,C1,M,_,PLT,method) → addObject(P2,C2,M,_,_,_,_,PLT,method)
81.	* deleteObject(P1,C1,M,_,_,attribute) → renameObject(P2,C2,X,_,_,attribute,M)
82.	* deleteObject(P1,C1,M,_,PLT,method) → renameObject(P2,C2,X,_,PLT,method,M)
83.	* renameObject(P1,C1,M,_,_,attribute,X) → addObject(P2,C2,M,_,_,DefType, AMode,_,attribute)
84.	* renameObject(P1,C1,M,_,PLT,method,X) → addObject(P2,C2,M,_,_,DefType, AMode,PLT,method)

** Note: Assume P1.C1 is one of the ancestor's of P2.C2*

A.2 Bi-Directional FGTs Sequential Dependencies

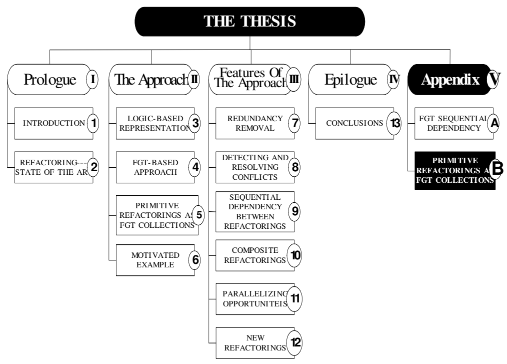
In the table below, all the bi-directional sequential dependencies between the different FGTs proposed in this thesis are catalogued. The information in the table is a continuation of the discussion in section 4.3.3. Each row in the table represents the following sequential dependencies: $FGT_x \leftrightarrow FGT_y$ where FGT_y is sequentially dependent on FGT_x and FGT_x is sequentially dependent on FGT_y. Note that each numbered row in the table corresponds to a numbered arc in Figure 4.1 that represents a bi-directional sequential dependency.

A.	deleteRelation(Ca,P,C,M,PR,PLT,Ftype,P1,C1,M1,PR1,PLT,Totype,Ltype) ↔ addRelation(Ca,P,C,M,PR, PLT,Ftype, P1,C1,M1,PR1,PLT,Totype,Ltype)
B.	renameObject(P,C,M,PR,PLT,parameter,X) ↔ renameObject(P,C,M,X,PLT,parameter,Y)
C.	renameObject(P,C,M,_,_,attribute,X) ↔ renameObject(P,C,X,_,_,attribute,Y)
D.	renameObject(P,C,M,_,PLT,method,X) ↔ renameObject(P,C,X,_,PLT,method,Y)
E.	renameObject(P,C,_,_,_,class,X) ↔ renameObject(P,X,_,_,_,class,Y)
F.	renameObject(P,C,M,PR,PLT,parameter,PR1) ↔ deleteObject(P,C,M,PR1,PLT,parameter)
G.	renameObject(P,C,M,_,_,attribute,M1) ↔ deleteObject(P,C,M1,_,_,attribute)
H.	renameObject(P,C,M,_,PLT,method,M1) ↔ deleteObject(P,C,M1,_,PLT,method)
I.	renameObject(P,C,_,_,_,class,M1) ↔ deleteObject(P,M1,_,_,_,class)
J.	renameObject(P,C,M,PR,PLT,parameter,PR1) ↔ addObject(P,C,M,PR,_,_,_,PLT,parameter)
K.	renameObject(P,C,M,_,_,attribute,PR1) ↔ addObject(P,C,M,_,_,_,_,_,attribute)
L.	renameObject(P,C,M,_,PLT,method,PR1) ↔ addObject(P,C,M,_,_,_,_,PLT,method)
M.	renameObject(P,C,_,_,_,class,PR1) ↔ addObject(P,C,_,_,_,_,_,_,class)
N.	addObject(P,C,M,PR,_,_,_,PLT,ObjT) ↔ deleteObject(P,C,M,PR,PLT,ObjT)
O.	addRelation(E,P,C,M,PR,PLT,ObjT, P1,C1,M1,PR1,PLT1,ObjT1,RT) ↔ renameRelation (E,P,C,M,PR, PLT,ObjT,P1,C1,M1,PR1,PLT1,ObjT1,RT,_)
P.	deleteRelation(E,P,C,M,PR,PLT,ObjT, P1,C1,M1,PR1,PLT1,ObjT1,RT) ↔ renameRelation (_,P,C,M,PR, PLT,ObjT, P1,C1,M1,PR1,PLT1,ObjT1,RT,E)
Q.	renameRelation (E1,P,C,M,PR,PLT,ObjT, P1,C1,M1,PR1,PLT1,ObjT1,RT,E2) ↔ renameRelation(E2,P,C, M,PR, PLT,ObjT,P1,C1,M1,PR1,PLT1,ObjT1,RT,E3)

Appendix B

PRIMITIVE REFACTORINGS AS FGT COLLECTIONS

This Appendix is a continuation of the discussion in chapter 5 that elaborates on the feasibility of representing primitive refactorings as a sequence of FGTs. Here we focus on primitive refactorings that map to a single FGT.


THE THESIS
Prologue I
The Approach II
Features Of The Approach III
Epilogue IV
Appendix V
INTRODUCTION 1
REFACTORING STATE OF THE ART 2
LOGIC-BASED REPRESENTATION 3
FGT-BASED APPROACH 4
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS 5
MOTIVATED EXAMPLE 6
REDUNDANCY REMOVAL 7
DETECTING AND RESOLVING CONFLICTS 8
SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS 9
COMPOSITE REFACTORINGS 10
PARALLELIZING OPPORTUNITEIS 11
NEW REFACTORINGS 12
CONCLUSIONS 13
FGT SEQUENTIAL DEPENDENCY A
PRIMITIVE REFACTORINGS AS FGT COLLECTIONS B

B.1 Add Element Refactorings

B.1.1 addClass(*ClassName, AccessMode*)

Where *ClassName* has the following format: *Pn.Cn* (*Pn* is the name of the package and *Cn* is the name of the class).

Description

The refactoring creates a new class *Cn* with access mode *AccessMode* in the package *Pn*, the created class will be empty and standalone (no members, super or subclasses)

Precondition Conjuncts

(1) The name of the new class *Cn* is distinct from those all classes declared already in the package *Pn*.

(2) The access mode for the new class is a valid access mode for classes.

FGT-List

1. addObject(*Pn, Cn, _, _, _, _, AccessMode, _, class*)

Note

Precondition conjuncts (1) and (2) are covered by precondition conjuncts of FGT 1 (*section 4.2.1.1.A*). There is no need to add precondition conjuncts at the refactoring-level.

B.1.2 addMethod(*MethodName, ReturnDType, AccessMode, ParameterList*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *ReturnDType* has the following format: *type(Type, Tname, Num)*
- *ParameterList* has the following format: [*(Prm_1,type($Type_1$,$Tname_1$,Num_1)), (Prm_2,type($Type_2$,$Tname_2$,Num_2)), …., (Prm_n,type($Type_n$,$Tname_n$,Num_n))*], where each item *(Prm_i, type($Type_i$, $Tname_i$, Num_i))* in the list represent information about a parameter defined in the method. The description of arguments of each item is as follows:
  - Prm_i is the name of the parameter.
  - $Type_i$ is the definition type of the parameter (*basic* or *complex*).
  - $Tname_i$ is the type name (*int, float,….*).
  - Num_i is the size of the array. (Zero if the parameter is not array).

Description

The refactoring creates a new method *Methn* with a list of parameters *ParameterList* in the class *Pn.Cn*. The method will have access mode *AccessMode* and return type *ReturnDType*.

Precondition Conjuncts

(1) The signature of the new method is distinct from those all methods declared already in the class *Pn.Cn* or any of its ancestors.

(2) Each parameter name is distinct from all other parameter's name in the parameter list *ParameterList*.

(3) The definition type of the return value of the method is valid and accessible.

(4) The access mode of the method is valid.

FGT-List

1. addObject(*Pn, Cn, Methn,_,_, ReturnDType, AccessMode , ParameterList, method*)

Note

Precondition conjuncts (1), (2), (3) and (4) are covered by precondition conjuncts of FGT 1 (*section 4.2.1.1.B*). There is no need to add precondition conjuncts at the refactoring-level.

B.1.3 addAttribute(*AttibuteName, AttributeDType, AccessMode*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *AttributeDType* has the following format: *type(Type, Tname, Num)*

Description

The refactoring creates a new attribute *Attn* in the class *Pn.Cn* with access mode *AccessMode*. The definition type of the new attribute will be *AttributeDType*.

Precondition Conjuncts

(1) The name of the new attribute is distinct from those all attributes declared already in the class *Pn.Cn* or any of its ancestors.

(2) The definition type of the attribute is valid and accessible.

(3) The access mode of the attribute is valid.

FGT-List

1. addObject(*Pn, Cn, Attn,_,_, AttributeDType, AccessMode ,_, attribute*)

Note

The precondition conjuncts (1), (2) and (3) are covered by the set of precondition conjuncts of the FGT 1 (*section 4.2.1.1.C*). There is no need to add precondition conjuncts at the refactoring-level.

B.1.4 addParameter(*Prmname, PrmDType, Index, MethTList*)

Where

- *Prmname* has the following format: *Pn.Cn.Methn.Prmn*
- *PrmDType* has the following format: *type(Type, Tname, Num)*
- *MethTList: [Tname$_1$, Tname$_2$,…., Tname$_n$]*, where each item *Tname$_i$* in the list represent the name of the definition type (*int, float,* ….) of one of the parameters defined in the method *Methn* in the same order as defined in the method. The list is used in addition to the name of the method to specify the signature of the method *Methn*.

Description

The refactoring declares a new parameter *Prmn* in the method *Pn.Cn.Methn* with *MethTList*. The type of the new parameter is defined by the variable *PrmDType*. The new parameter will be added at the index *Index* of the list of the method parameters. If that *Index* is occupied then all the parameters from the *Index* will be shifted one-step to the right

Precondition Conjuncts

(1) The parameter name is distinct in the method's parameters list.

(2) The produced method signature muse be distinct from all those methods define in the class *Pn.Cn* or any of its ancestors.

(3) The parameter definition type should be valid and accessible.

FGT-List

1. addObject(*Pn, Cn, Methn, Prmn, Index, PrmDType,_,MethTList, parameter*)

Note

The precondition conjuncts (1), (2) and (3) are covered by the set of precondition conjuncts of the FGT 1 (*section 4.2.1.1.D*). There is no need to add precondition conjuncts at the refactoring-level.

B.2 Rename Element Refactorings

B.2.1 renameClass(*ClassName, NewName*)

Where *ClassName* has the following format: *Pn.Cn*

Description

The refactoring changes the name of the class *Pn.Cn* to a new name *Pn.NewName*. The renameClass refactoring is a behaviour-preserving refactoring because changing the name of the class will not have any effect on the behaviour of the system.

Precondition Conjuncts

(1) The new name of the class *NewName* should not clash with any other class names declared in the package *Pn*.

FGT-List

1. renameObject(*Pn, Cn, _, _, _, class, NewName*)

Note

- Precondition conjunct (1) is covered by precondition conjuncts of the FGT 1 (*section 4.2.1.2.A*). There is no need to add precondition conjuncts at the refactoring-level.

B.2.2 renameMethod(*MethodName, MethTList, NewName*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[*$Tname_1$*,* $Tname_2$*,...., *$Tname_n$*]*

Description

The refactoring changes the name of the method *Methn* with parameter list *MethTList* defined in the class *Pn.Cn* to another name *Pn.Cn.NewName*. The renameMethod refactoring is a behaviour-preserving refactoring because changing the name of the method will not have any effect on the behaviour of the system.

Precondition Conjuncts

(1) The signature of the method with the new name should not clash with the signature of other methods declared in the class *Pn.Cn* or any of its ancestors.

FGT-List

1. renameObject(*Pn, Cn, Methn,_, MethTList, method, NewName*)

Note

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 (*section 4.2.1.2.B*). There is no need to add precondition conjuncts at the refactoring-level.

B.2.3 renameAttribute(*AttributeName, NewName*)

Where *AttributeName*: *Pn.Cn.Attn*

Description

The refactoring changes the name of the attribute *Attn* declared in the class *Pn.Cn* to another name *Pn.Cn.NewName*. The renameAttribute refactoring is a behaviour-preserving refactoring because changing the name of the attribute will not have any effect on the behaviour of the system.

Precondition Conjuncts

(1) The new name of the attribute *NewName* should not clash with any other attributes names declared in the class *Pn.Cn* or any of its ancestors.

FGT-List

1. renameObject(*Pn, Cn, Attn, _, _, attribute, NewName*)

Note

A precondition (1) is covered by precondition conjuncts of FGT 1 (*section 4.2.1.2.C*). There is no need to add precondition conjuncts at the refactoring-level.

B.2.4 renameParameter(*ParameterName, MethTList, NewName*)

Where

- *ParameterName* has the following format: *Pn.Cn.Methn.Prmn*
- *MethTList* has the following format: *[$Tname_1$, $Tname_2$,…., $Tname_n$]*

Description

The refactoring changes the name of the parameter *Prmn* declared in the method *Methn* with parameter list *MethTList* to another name *NewName*. The renameParameter refactoring is a behaviour-preserving refactoring because changing the name of the parameter will not have any effect on the behaviour of the system.

Precondition Conjuncts

(1) The parameter's new name should not clash with the names of those parameters that are declared in the method *Pn.Cn.Methn* with *MethTList*.

FGT-List

1. renameObject(*Pn, Cn, Methn, Prmn, MethTList, parameter, NewName*)

Note

Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 (*section 4.2.1.2.D*). There is no need to add precondition conjuncts at the refactoring-level.

B.3 Change Characteristics Refactorings

B.3.1 changeClassAccess(*ClassName, NewAcces*)

Where *ClassName* has the following format: *Pn.Cn*

Description

The refactoring changes the class *ClassName* access mode.

Precondition Conjuncts

(1) In the case of changing the access mode of the class *Cn* from a lower restriction access mode to a higher restriction one, all the references made by other *object* elements in the system to the class before the refactoring should be within the scope of the class after the refactoring. Since changing the access mode of the class does not affect any of the references to it, this refactoring will not change the behaviour of the system.

FGT-List

1. changeOAMode(*Pn, Cn, _, _, _, class, OOldAM, NewAcces*)

Note

Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 (*section 4.2.1.3.A*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current access mode of the class we use the procedure **objectAMode**(*Pn, Cn, class, OOldAM*).

B.3.2 changeMethodAccess(*Methname, MethTList, NewAccess*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname*$_1$*, Tname*$_2$*,…., Tname*$_n$*]*

Description

The refactoring changes the access mode of the method.

Precondition Conjuncts

(1) In the case of changing the access mode of the method *Methn* from a lower restriction access mode to a higher restriction one, all the references made by other *object* elements in the system to the method before the refactoring should be within the scope of the method after the refactoring. Since changing the access mode of the method does not affect any of the references to it, this refactoring will not change the behaviour of the system.

FGT-List

1. changeOAMode(*Pn, Cn, Methn,_, MethTList, method, OOldAM, NewAccess*)

Note

The precondition of this refactoring is covered by precondition conjuncts of FGT 1 (*section 4.2.1.3.B*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current access mode of the method we use the procedure **objectAMode**(*Pn, Cn, Methn, MethTList, method, OOldAM*).

B.3.3 changeAttributeAccess(*AttributeName, NewAccess*)

Where *AttributeName* has the following format: *Pn.Cn.Attn*

Description

The refactoring changes the access mode of the attribute.

Precondition Conjuncts

(1) In the case of changing the access mode of the attribute *Attn* from a lower restriction access mode to a higher restriction one, all the references made by other *object* elements in the system to the attribute before the refactoring should be within the scope of the attribute after the refactoring. Since changing the access mode of the attribute does not affect any of the references to it, this refactoring will not change the behaviour of the system.

FGT-List

1. changeOAMode(*Pn, Cn, Attn,_,_, attribute, OOldAM, NewAccess*)

Note

The precondition of this refactoring is covered by precondition conjuncts of FGT 1 (*section 4.2.1.3.C*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current access mode of the attribute we use the procedure **objectAMode**(*Pn, Cn, Attn, OOldAM*).

B.3.4 changeMethodReturnType(*Methodname, MethTList, NewRType*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[*$Tname_1$*,* $Tname_2$*,…., *$Tname_n$*]*
- *NewRType* has the following format: *type(Type, Tname, Num)*

Description

The refactoring changes the definition type of the return value of the method.

Precondition Conjuncts

(1) The method *Pn.Cn.Methn* with *MethTList* should be defined in the system.

(2) The *NewRType* should be valid and accessible.

FGT-List

1. changeODefType(*Pn, Cn, Methn,_, MethTList, method, OldRType, NewRType*)

Note

Precondition conjuncts (1) and (2) of this refactoring are covered by precondition conjuncts of FGT 1 (*section 4.2.1.4.A*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current definition type of the return value of the method we use the procedure **objectDType**(*Pn, Cn, Methn, MethTList, method, OldRType*).

B.3.5 changeAttributeDefType(*AttributeName, NewDType*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *NewDType* has the following format: *type(Type, Tname, Num)*

Description

The refactoring changes the definition type of the attribute.

Precondition Conjuncts

(1) The attribute *Pn.Cn.Attn* should be defined in the system.

(2) The *NewDType* should be valid and accessible.

FGT-List

1. changeODefType(*Pn, Cn, Attn,_, _, attribute, OldDType, NewDType*)

Note

Precondition conjuncts (1) and (2) of this refactoring are covered by precondition conjuncts of FGT 1 (*section 4.2.1.4.B*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current definition type of the return value of the method we use the procedure **objectDType**(*Pn, Cn, Attn, attribute, OldDType*).

B.3.6 changeParameterDefType(*Parametername, MethTList, NewDType*)

Where

- *ParameterName* has the following format: *Pn.Cn.Methn.Prmn*
- *MethTList* has the following format: *[*$Tname_1$*,* $Tname_2$*,…., *$Tname_n$*]*
- *NewDType* has the following format: *type(Type, Tname, Num)*

Description

The refactoring changes the definition type of one of the parameters of the *Methn*.

Precondition Conjuncts

(1) The parameter *Prmn* should be declared in the method *Pn.Cn.Methn* with *MethTList*.

(2) The *NewDType* should be valid and accessible.

FGT-List

1. changeODefType(*Pn, Cn, Methn,Prmn, MethTList, parameter, OldDType, NewDType*)

Note

Precondition conjuncts (1) and (2) of this refactoring are covered by precondition conjuncts of FGT 1 (*section 4.2.1.4.C*). There is no need to add precondition conjuncts at the refactoring-level. To retrieve the current definition type of the return value of the method use the procedure **objectDType**(*Pn, Cn, Methn, Prmn, MethTList, parameter, OldDType*).

B.4 Delete Element Refactorings

B.4.1 deleteMethod(*MethodName, MethTList*)

Where

- *MethodName* has the following format: *Pn,Cn, Methn*
- *MethTList* has the following format: *[*$Tname_1$*,* $Tname_2$*,…., *$Tname_n$*]*

Description

The refactoring deletes unreferenced method *Methn* with the parameter list *MethTList* from the class *Pn.Cn*

Precondition Conjuncts

(1) The method *Methn* with the parameter list *MethTList* should be declared in the class *Pn.Cn*.

(2) The method is unreferenced by any other *object* elements.

(3) If the method is inherited by subclasses of the class *Pn.Cn* then the method also should be unreferenced by any instances of these classes.

FGT-List

1. deleteObject(*Pn, Cn, Methn, _, MethTList, method*)

Note

- Precondition conjuncts (1), (2) and (3) are covered by the set of precondition conjuncts of the FGT 1 (*section 4.2.1.1.B*). There is no need to add precondition conjuncts at the refactoring-level of this refactoring.

B.4.2 deleteAttribute(*AttributeName*)

Where *AttributeName* has the following format: *Pn.Cn.Attn*

Description

The refactoring deletes unreferenced attribute *Attn* from the class *Pn.Cn*.

Precondition Conjuncts

(1) The attribute *Attn* should be declared in the class *Pn.Cn*.

(2) The attribute is unreferenced by any other *object* elements.

(3) If the attribute Attn is inherited by subclasses of the class *Pn.Cn* then the attribute *Attn* should not be referenced by any instances of these classes.

FGT-List

1. deleteObject(*Pn, Cn, Attn,_,_,attribute*)

Note

The precondition conjuncts (1), (2) and (3) are covered by the set of precondition conjuncts of the FGT 1 (*section 4.2.1.5.C*). There is no need to add precondition conjuncts at the refactoring-level.

B.4.3 deleteParameter(*Prmname, MethTList*)

Where

- *Prmname* has the following format: *Pn.Cn.Methn.Prmn*
- *MethTList* has the following format: *[$Tname_1$, $Tname_2$,…., $Tname_n$]*

Description

The refactoring removes the parameter *Prmn* from the parameter's list of the method *Methn*. This refactoring is beneficial when, for example, a method's purpose is changed and there is a need to remove (and perhaps later add) parameters from the method.

Precondition Conjuncts

(1) The parameter should be declared in the method.

(2) The produced method signature after removing the parameter should not be declared in the class *Pn.Cn* or in any of its ancestors.

FGT-List

1. deleteObject(*Pn, Cn, Methn, Prmn, MethTList, parameter*)

Note

The precondition conjuncts (1) and (2) are covered by the set of precondition conjuncts of the FGT 1 (section 4.2.1.5.D). There is no need to add precondition conjuncts at the refactoring-level.

BIBLIOGRAPHY

[1] Arnold, R. (1986). An introduction to software restructuring. *In Tutorial on Software Restructuring*, Robert S. Arnold, Ed. IEEE.

[2] Astels, D. (2002). Refactoring with UML. *In Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering* (pp. 67-70).

[3] Back, R. (2002). Software Construction by Stepwise Feature Introduction. *In Didier Bert et. Al., editor, ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of Lecture Notes in Computer Science, pages 162-183.

[4] Banerjee, J., and Kim, W. (1987). Semantics and implementation of schema evolution in object-oriented databases. *In Proc. SIGMOD Conf.*, ACM.

[5] Binkley, D. and Gallagher, K. (1996). Program slicing. *Advances of Computing 43*, pp. 150.

[6] Boger, P., Sturm, T., and Fragemann, P. (2002). Refactoring Browser for UML. *In Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, pages 77-81, Alghero, Sardinia, Italy.

[7] Bottoni, P., Parisi-Presicce, F., and Taentzer, G. (2002). Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science* 72(4).

[8] Cinne'ide, M. (2000). *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, Department of Computer Science – Trinity College – Dublin.

[9] Coleman, D. , Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *IEEE Computer*, vol. 27, no. 8, pp. 44–49.

[10] Corradini, A., Ehrig, H., Kreowski, H., and Rozenberg, G. (2002). Graph Transformation. *Lecture Notes in Computer Science 2505*, Springer-Verlag.

[11] Cuny, J., Ehrig, H., Engels, G., and Rozenberg, G. (1996). Graph Grammars and Their Application to Computer Science. *Lecture Notes in Computer Science* 1073, Springer-Verlag.

[12] D'Hondt, M. (1998). *Managing Evolution of Changing Software Requirements*. Dissertation, Department of Computer Science, Vrije Universiteit Brussel.

[13] Demeyer, S. *et al*. (2005). The LAN-simulation: A Refactoring Teaching Example. *Int. Workshop on Principles of Software Evolution (IWPSE)*. pages 123-134.

[14] Deursen, V., and Kuipers, T. (1999). Identifying objects using clusterand concept analysis. *In Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 246–255. IEEE Computer Society.

[15] Ecklund E., Lois, M., and Freiling, M. (1996). Change cases: Use cases that identify future requirements. *Proceedings of OOPSLA '96, ACM SIGPLAN Notices*, 31(10), pp. 342-358, ACM Press.

[16] Edwards, W. (1997). Flexible Conflict Detection and Management in Collaborative Applications. *Proc. Symp. User Interface Software and Technology*.

[17] Eetvelde, N., and Janssens, D. (2003). A hierarchical program representation for refactoring. *In Proc. of UniGra'03 Workshop*.

[18] Ehrig, H., Engels, G., Kreowski, J., & Rozenberg, G. (2000). Theory and Application to Graph Transformations. *Lecture Notes in Computer Science* 1764,Springer-Verlag.

[19] Engels, G., Hartmut, E., & Rozenberg, G. (1996). Special Issue on Graph Transformations. *Fundamenta Informaticae* 26 (3,4), IOS Press.

[20] Fanta, R. and Rajlich, V. (1999). Restructuring legacy C code into C++. *In Proc. Int'l Conf. Software Maintenance*, pp. 77-85.

[21] Feather, M. (1989). Detecting Interference when Merging Specification Evaluations. *Proc. Fifth Int'l Workshop Software Specification and Design*, pp. 169-176.

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

[23] France, R., and James M., (2001). Multi-View Software Evolution: A UML based Framework for Evolving Object-Oriented Software. *In Proceedings of The IEEE Interna- tional Conference on Software Maintenance*.

[24] Ganter, B., and Wille, R. (1999). Formal Concept Analysis: mathematical foundations. *(Translated from the German by Cornelia Franzke) Springer-Verlag*, Berlin-Heidelberg.

[25] Glass, R. (1998). Maintenance: Less is not more. *IEEE Software* 15(4): 67-68.

[26] Gorp, P., Stenten, H., Mens, T., and Demeyer, S. (2003). Towards automating source consistent UML refactorings. *In Proc. UML 2003, vol. 2863 of Lecture Notes in Computer Science*, pp. 144–158, Springer-Verlag.

[27] Griswold, W. (1991). *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington.

[28] Griswold, W., and Notkin, D. (1993). Automated assistance for program restructuring. *Trans. Software Engineering and Methodology*, vol. 2, no. 3, pp. 228–269, ACM.

[29] Guimaraes, T. (1983). Managing application program maintenance expenditure. *Comm. ACM*, vol. 26, no. 10, pp. 739–746.

[30] Hannemann, J., and Kiczales, G. (2001). Overcoming the prevalent decomposition of legacy code. *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE), Toronto*. http://www.cs.ubc.ca/jan/papers/ICSE2001-OvercomingDecomposition.pdf .

[31] Heckel, R. (1995). *Algebraic graph transformations with application conditions*. M.S. thesis, TU Berlin.

[32] Hunt, J., and McIllroy, M. (1976). An Algorithm for Differential File Comparison. *Technical Report 41*, AT&T Bell Laboratories Inc.

[33] Jacobson, I., Christerson, M., Jonsson , P., and Övergaard, G. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

[34] Jahnke, J., and Zundorf, A. (1997). Rewriting poor design patterns by good design patterns. *In S. Demeyer and H. Gall,editors. Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering*, Technical University of Vienna,Technical Report TUV-1841-97-10.

[35] JTransformer homepage http://roots.iai.uni-bonn.de/research/jtransformer/

[36] Kempen, M., Chaudron, M., Kourie, D., and Boake, A. (2005). Towards proving preservation of behaviour of refactoring of UML models. *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT; Vol. 150, South Africa

[37] Kerievsky, J. (2002). *Refactoring To Patterns*. Technical report, Industrial Logic. http://www.industriallogic.com/xp/ refactoring/.

[38] Kniesel, G. & Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming*, 52:9-51.

[39] Kniesel, G. (2006). A logic foundation for conditional program transformations. *Technical report no IAI-TR-2006-01*,ISSN 0944-8535,CS Dept III.

[40] Komondoor, R. and Horwitz, S. (2000). Semantics-preserving procedure extraction. *Technical report, Computer Sciences Department*, University of Wisconsin Madison.

[41] Kramer, J., and Magee J., (1998). Analysing Dynamic Change in Software Architectures. *Proceedings of IWPSE98, International Workshop on Principles of Software Evolution*, Kyoto, Japan.

[42] Lakhotia, A. and Deprez, J. (1998). Restructuring programs by tucking statements into functions. *In: M. Harman and K. Gallagher, editors, Special Issue on Program Slicing, Information and Software Technology 40, Elsevier*, pp. 677-689.

[43] Lanubile, F., and Visaggio, G., (1997). Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering* 23 4, pp. 246–259.

[44] Leblang, D. and Chase, R. (1984). Computer-Aided Software Engineering in a Distributed Workstation Environment. *Proc. SIGPLAN/SIGSOFT Software Eng. Symp. Practical Software Development Environments*, ACM SIGPLAN Notices, vol. 19, no. 5, pp. 104-112.

[45] Leblang, D., Chase, R. and Spilke, H. (1988). Increasing Productivity with a Parallel Configuration Manager. *Proc. Int'l Workshop Software Version and Configuration Control*. pp. 21-38.

[46] Lehman, M., Belady, L. (1985). Program Evolution, *P*rocesses of software change. *Academic Press Professional, Inc.,* San Diego, CA.

[47] Lientz, B., and Swanson, E. (1980). *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*, Addison-Wesley.

[48] Lubkin, D. (1991). Heterogeneous Configuration Management with DSEE. *Proc. Third Int'l Workshop Software Configuration Management*, pp. 153-160.

[49] Markovic', S. (2004). Composition of UML Described Refactoring Rules. *OCL and Model Driven Engineering, UML 2004 Conference Workshop*. Lisbon, Portugal, Octavian Patrascoiu (Ed.), University of Kent, pp. 45-59.

[50] Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering,* vol. 28 n.5, p. 449-462.

[51] Mens, T. (1999). *A Formal Foundation for Object-Oriented Software Evolution.* PhD Thesis, Dept. Computer Science, Vriji Univ. Brussel, Belgium.

[52] Mens, T. (2001). Transformational Software Evolution by Assertions. Formal Foundations for the Evolution of Hypermedia Systems. *5th European Conference on Software Maintenance and Reengineering, Workshop on FFSE.* IEEE Press. Lisbon, Portugal, pp. 67-74.

[53] Mens, T. (2005). On the use of graph transformations for model refactoring. *In Generative and transformational techniques in software engineering* (J. V. Ralf Lämmel,Joao Saraivaed.), pp. 67–98, Departamento di Informatica,Universidade do Minho.

[54] Mens, T., Demeyer, S., and Janssens, D. (2002). Formalising behaviour preserving program transformations. *In Graph Transformation, Lecture Notes in Computer Science,* vol. 2505, pp. 286-301, Springer-Verlag.

[55] Mens, T., Kniesel, G., and Runge, O. (2006). Transformation dependency analyis: A comparison of two approaches. *Proceedings of Langages et Modèles à Objects* (LMO2006).

[56] Mens, T., Lucas, P., and Steyaert, P. (1998). Supporting reuse and evolution of UML models. *In P.-A. Muller and J. Bézivin, editors, Proceeding of <<UML>>'98 International Workshop*, Mulhouse, France, pages 341-350.

[57] Mens, T., Taentzer, G., and Runge, O. (2005). Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, vol. 127, n° 3, p. 113-128

[58] Mens, T., Taentzer, G., and Runge, O. (2006). Analyzing Refactoring Dependencies Using Graph Transformation. *Software and Systems Modeling*.

[59] Mens, T., Tourwe', T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering,* vol. 30 n.2, p. 126-139.

[60] Mens, T., Van Eetvelde, N., Demeyer, S., & Janssens, D. (2005). Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution* 17(4), 247–276, Wiley.

[61] Mens, T., Van Gorp, P., ,Varró, D., & Karsai, G. (2005). Applying a model transformation taxonomy to graph transformation technology. *In Proc. Int'l* (GraMoT2005).

[62] Munson, J. and Dewan, P. (1994). A Flexible Object Merging Frame-work. *Proc. ACM Conf. Computer Supported Collaborative Work*, pp. 231-241.

[63] Nagl, M., Sch¨urr, A., and M¨unch, M. (2000). Applications of Graph Transformations with Industrial Relevance. *Lecture Notes in Computer Science,* volume 1779. Springer-Verlag.

[64] Object Management Group (2005). Unified Modeling Language: Infrastructure version 2.0. *Formal/2005-07-05.*

[65] Opdyke, W. (1992). *Refactoring object-oriented frameworks*. Ph.D. thesis. University of Illinois at Urbana-Champaign.

[66] Opdyke, W., & Johnson R. (1993). Creating abstract superclasses by refactoring. *Proceedings ACM Computer Science Conference. ACM Press*, pp. 66-73.

[67] Philipps, J. and Rumpe, B. (1997). Refinement of information flow architectures. *In Proc. ICFEM'97*. IEEE Computer Society.

[68] Porres, I. (2003). Model refactorings as rule-based update transformations. *Proceedings of UML 2003 Conference*, pages 159-174.

[69] Proceedings of International Workshop on Principles of Software Evolution, Kyoto, Japan, 1998. ACM SIG Publication, ACM Press, 1999.

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

[71] Roberts, D., Brant, J., & Johnson, R. (1997). A refactoring tool for smalltalk. *Theory and Practice of Object Systems,* vol. 3, no. 4, pp. 253-263.

[72] Russo, A., Nuseibeh, B., and Kramer, J. (1998). Restructuring requirements specifications for managing inconsistency and change: A case study. *In Proc. Int'l Conf. Requirements Engineering*, pp. 51-61, Colorado Spring, USA.

[73] Saadeh, E., Kourie, D. & Boake, A. (2008). Model Refactorings as Logic-Based Fine-Grain Transformations. *Proceedings of the 9th African Conference on Research in Computer Science and Applied Mathematics*, 703-710, ISBN: 2-7261-1299-4, Rabat, Morocco.

[74] Saadeh, E., Kourie, D., & Boake, A. (2008). An Algorithm for Ordering Refactorings Based on Fine-Grained Model Transformations. *7th International Conference on Software Methodologies, Tools and Techniques*, 225-243,ISSN:0922-6389,SOMET08, Sharjah, UAE.

[75] Saadeh, E., Kourie, D., and Boake, A. (2009). Fine-grain Transformations to Refactor UML Models. *Proceedings of The Warm Up Workshop for ACM/IEEE ICSE 2010*, 45-51, ACM ISBN: 978-1-60558-565-9, Cape Town, South Africa.

[76] Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics based refactoring. *Proc European Conf Software Maintenance and Reengineering*, pages 30-38.

[77] Snelting, G., and Tip, F. (1998). Reengineering class hierarchies using concept analysis. *In Proc. Foundations of Software Engineering (FSE-6), SIGSOFT Software Engineering Notes 23(6)*, pp. 99-110.

[78] Soley, R. (2000). Model Driven Architecture. *OMG Document omg*.

[79] Sommerville, (2004). *Software evolution and reengineering*. Software Engineering 7th Edition (Chp 21) Addison-Wesley.

[80] Steyaert, P., Lucas, C., Mens, K. and D'Hondt, T. (1996). Reuse Contracts: Managing the Evolution of Reusable Assets. *Proc. OOPSLA '96, ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 268-286.

[81] Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). Refactoring UML models. *In Proc. Int'l Conf. Unified Modeling Language* (pp. 134-138), LNCS 2185, Springer.

[82] Tip, F.(1995). A survey of program slicing techniques. *Journal of Programming Languages 3(3)*, pp. 121-189.

[83] Tokuda, L., and Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engineering*, vol. 8, no.1, pp. 89–120.

[84] Tonella, P. (2001). Concept analysis for module restructuring. *Trans. Software Engineering 27(4)*, pp. 351-363.

[85] Tourwe, T. and Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. *Proc 7th European Conf Software Maintenance and Re-engineering* (CSMR2003), IEEE Computer Society Press, pages 91-100.

[86] Ward, M., and Bennett, K. (1995). Formal methods to aid the evolution of software. *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. 5, no. 1, pp. 25–47.

[87] Wermelinger, M. (1998). Software architecture evolution and the chemical abstract machine. *In International Workshop on the Principles of Software Evolution*, pages 93–97, Kyoto, Japan.

[88] Wiels, V., and Easterbrook, S. (1998). Management of Evolving Specifications Using Category Theory. *Proceedings of Automated Software Engineering Conference '98*, pp. 1221, IEEE Press.