CLASSIFICATION OF REFACTORING METHODS BASED ON SOFTWARE QUALITY ATTRIBUTES

BY

KARIM OMAR ELISH

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June 2008

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261. SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **KARIM OMAR ELISH** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

Thesis Committee

Dr. Mohammad Alshayeb (Thesis Advisor)

Dr. Mahmoud Elish (Member)

Dr. Muhammed Al-Mulhem (Member)

Dr. Kanaan A. Faisal
(Department Chairman)

Dr. Mohammad Al-Homoud
(Dean of Graduate Studies)

9/8/08
Date

Dedication

To my beloved family:

my parents, my brothers (Mahmoud and Sameh), my nephews (Omar, Ali, and Adam), and my lovely fiancée.

Acknowledgements

I would like to take this chance to acknowledge King Fahd University of Petroleum and Minerals (KFUPM) for all support extended during this research.

Furthermore, I would like to express my gratitude and appreciation to my thesis advisor, Dr. Mohammad Alshayeb, for his continuous support, unlimited help, valuable guidance and advice since we first met. My gratitude is also due to the thesis committee members, Dr. Mahmoud Elish and Dr. Muhammed Al-Mulhem for their help and enlightening comments.

Thanks to all colleagues and friends for their suggestions and encouragement during the preparation of this thesis.

Last, but not least, I would like to thank my parents, my brothers, my nephews, and my fiancée for their prayers, encouragement, and continuous support.

Table of Contents

Page

Acknowledgements……...iv

List of Tables viii

List of Figures..... xii

Abstract (English) xiv

Abstract (Arabic) xv

Chapter 1. Introduction..... 1

1.1. Problem Statement 2
1.2. Rationale: Problem Importance 3
1.3. Research Contributions 4
1.4. Thesis Organization 5

Chapter 2. Background 6

2.1. Software Refactoring 6
2.1.1. Definition of Refactoring 6
2.1.2. Benefits of Refactoring 7
2.1.3. Refactoring Catalog 8
2.1.4. Refactoring Methods under Study 9
2.2. Software Refactoring to Patterns 21
2.2.1. Definition of Refactoring to Patterns 21
2.2.2. Refactoring to Patterns Catalog 22
2.2.3. Refactoring to Patterns under Study 22
2.3. Software Quality Attributes 32
2.3.1. Internal Quality Attributes 32
2.3.2. External Quality Attributes 33

Chapter 3. Literature Review 35

3.1. Assessing Refactoring Effect on Internal Software Quality 35
3.2. Assessing Refactoring Effect on External Software Quality 37
3.3. Mapping Refactoring Effect on Internal Software Quality to External Software Quality 38

Chapter 4. Relating Internal Software Quality to External Software Quality 42

4.1. Internal Quality Attributes under Study 43
4.2. External Quality Attributes under Study 45
4.3. External Quality Attributes Assessments Using Internal Quality Attributes 46
4.3.1. Adaptability, Completeness, Maintainability, Understandability, and Reusability Assessments Using Internal Quality Metrics 47
4.3.2. Testability Assessment Using Internal Quality Metrics 47
4.3.3. Reliability Assessment Using Internal Quality Metrics 48

Chapter 5. Refactoring Classification Based on Software Quality Attributes 52

5.1. Refactoring Methods Classification 52
5.1.1. Selected Refactoring Methods 53
5.1.2. Source Code Examples 54
5.1.3. Classification of Refactoring Methods Based on Internal Quality Attributes 67
5.1.4. Classification of Refactoring Methods Based on External Quality Attributes 75
5.2. Refactoring to Patterns Classification 78
5.2.1. Selected Refactoring to Patterns 78
5.2.2. Source Code Examples 79
5.2.3. Classification of Refactoring to Patterns Based on Internal Quality Attributes 91
5.2.4. Classification of Refactoring to Patterns Based on External Quality Attributes 99

Chapter 6. Empirical Validations 101

6.1. Case Study I: Validating Refactoring Methods Classification 101
6.1.1. Software Systems Background 102
6.1.2. Data Collection 103
6.1.3. Results from Course Project 01 106
6.1.4. Results from Course Project 02 111
6.1.5. Results from Course Project 03 115
6.1.6. Results from JLOC 119

6.1.7. Results from J2Sharp 122
6.1.8. Results from JNFS 125
6.1.9. Discussion of Results 129
6.2. Case Study II: Validating Refactoring to Patterns Classification 142
6.2.1. Software Systems Background 142
6.2.2. Data Collection 143
6.2.3. Results from HTML Parser 143
6.2.4. Results from Java Class Browser 146
6.2.5. Results from Java Neural Network Trainer 148
6.2.6. Results from JMK 150
6.2.7. Discussion of Results 153

Chapter 7. Conclusion 158

7.1. Major Contributions 158
7.2. Future Work 159

References 161

Vita 166

List of Tables

Table Page

Table 3-1. Summary of related work 41

Table 4-2. The relationship between internal and external software quality attributes 51

Table 5-3. Measurement results for source code examples before and after applying refactoring methods 68

Table 5-4. Refactoring methods classification based on internal software quality attributes..... 74

Table 5-5. Refactoring methods classification based on external software quality attributes..... 77

Table 5-6. Measurement results for source code examples before and after applying refactoring to patterns 92

Table 5-7. Refactoring to patterns classification based on internal software quality attributes..... 98

Table 5-8. Refactoring to patterns classification based on external software quality attributes..... 100

Table 6-9. The characteristics of studied software projects in case study I..... 102

Table 6-10. Number of applied refactoring methods on each software project in case study I..... 105

Table 6-11. Measurement results before applying any refactoring method: Course Project 01 106

Table 6-12. Measurement results for the affected classes before and after applying refactoring methods: Course Project 01 107

Table 6-13. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 01 110

Table 6-14. Measurement results before applying any refactoring method: Course Project 02..... 111

Table 6-15. Measurement results for the affected classes before and after applying refactoring methods: Course Project 02..... 112

Table 6-16. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 02 114

Table 6-17. Measurement results before applying any refactoring method: Course Project 03..... 115

Table 6-18. Measurement results for the affected classes before and after applying refactoring methods: Course Project 03..... 116

Table 6-19. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 03 118

Table 6-20. Measurement results before applying any refactoring method: JLOC..... 119

Table 6-21. Measurement results for the affected classes before and after applying refactoring methods: JLOC..... 120

Table 6-22. Changes in the internal quality metrics caused by applying refactoring methods: JLOC 121

Table 6-23. Measurement results before applying any refactoring method: J2Sharp 122

Table 6-24. Measurement results for the affected classes before and after applying refactoring methods: J2Sharp 123

Table 6-25. Changes in the internal quality metrics caused by applying refactoring methods: J2Sharp..... 124

Table 6-26. Measurement results before applying any refactoring method: JNFS 125

Table 6-27. Measurement results for the affected classes before and after applying refactoring methods: JNFS 126

Table 6-28. Changes in the internal quality metrics caused by applying refactoring methods: JNFS 128

Table 6-29. Classification of refactoring methods based on internal software quality attributes using empirical results 130

Table 6-30. Classification of refactoring methods based on external software quality attributes using empirical results 131

Table 6-31. Refactoring methods that have inverse effect on the internal quality metrics 135

Table 6-32. Sum of measurement results before and after applying group of refactoring methods 136

Table 6-33. Average of measurement results before and after applying group of refactoring methods 137

Table 6-34. Changes in the internal quality metrics caused by applying group of refactoring methods 138

Table 6-35. The effect of refactoring groups on the external quality attributes 141

Table 6-36. The characteristics of studied software projects in case study II 142

Table 6-37. Measurement results for lexer package: HTML Parser 143

Table 6-38. Measurement results for tests package: HTML Parser 144

Table 6-39. Measurement results for the affected classes before and after applying refactoring to patterns: HTML Parser 145

Table 6-40. Measurement results before applying any refactoring to patterns: Java Class Browser 146

Table 6-41. Measurement results for the affected classes before and after applying refactoring to patterns: Java Class Browser 147

Table 6-42. Measurement results before applying any refactoring to patterns: Java Neural Network Trainer 148

Table 6-43. Measurement results for the affected classes before and after applying refactoring to patterns: Java Neural Network Trainer 149

Table 6-44. Measurement results for jmk package: JMK 150

Table 6-45. Measurement results for the affected classes before and after applying refactoring to patterns: JMK 152

Table 6-46. Classification of refactoring to patterns based on internal software quality attributes using empirical results 154

Table 6-47. Classification of refactoring to patterns based on external software quality attributes using empirical results 155

List of Figures

Figure Page

Figure 2-1. Consolidate conditional expression: before and after 10

Figure 2-2. Encapsulate field: before and after 11

Figure 2-3. Extract class: before and after 12

Figure 2-4. Extract method: before and after 13

Figure 2-5. Hide method: before and after 14

Figure 2-6. Inline class: before and after 15

Figure 2-7. Inline method: before and after 16

Figure 2-8. Inline temp: before and after 17

Figure 2-9. Remove setting method: before and after 18

Figure 2-10. Replace assignment with initialization: before and after 19

Figure 2-11. Replace magic number with symbolic constant: before and after 20

Figure 2-12. Reverse conditional: before and after 21

Figure 2-13. Chain constructors: before and after 24

Figure 2-14. Compose method: before and after 25

Figure 2-15. Form template method: before and after 27

Figure 2-16. Introduce null object: before and after 28

Figure 2-17. Replace conditional dispatcher with command: before and after 29

Figure 2-18. Replace constructors with creation methods: before and after 30

Figure 2-19. Unify interfaces: before and after 31

Figure 5-20. Consolidate conditional expression: source code example 55

Figure 5-21. Encapsulate field: source code example 56

Figure 5-22. Extract class: source code example 57

Figure 5-23. Extract method: source code example 58

Figure 5-24. Hide method: source code example 59

Figure 5-25. Inline class: source code example 60

Figure 5-26. Inline method: source code example 61

Figure 5-27. Inline temp: source code example 62

Figure 5-28. Remove setting method: source code example 63

Figure 5-29. Replace assignment with initialization: source code example 64

Figure 5-30. Replace magic number with symbolic constant: source code example 65

Figure 5-31. Reverse conditional: source code example 66

Figure 5-32. Chain constructors: source code example 81

Figure 5-33. Compose method: source code example 82

Figure 5-34. Form template method: source code example 83

Figure 5-35. Introduce null object: source code example 85

Figure 5-36. Replace conditional dispatcher with command: source code example 87

Figure 5-37. Replace constructors with creation methods: source code example 89

Figure 5-38. Unify interfaces: source code example 90

Abstract

Name: Karim Omar Elish

Title: Classification of Refactoring Methods Based on Software Quality Attributes

Major Field: Computer Science

Date of Degree: June 2008

One of the most commonly used techniques for improving software quality is called refactoring. Refactoring is the process of improving the design of existing code by changing its internal structure without affecting its external behavior. When applying refactoring methods, some quality attributes can be improved and some others can be impaired. This means that improving one quality attribute may affect negatively other quality attributes. However, there are no guidelines to help the software designer decide which refactoring methods to apply in order to optimize a software system with regard to certain design goals. In this thesis, we propose a classification of refactoring methods including refactoring to patterns based on their measurable effect on software quality attributes. Additionally, we empirically validate this classification by using real software systems. This study, in turn, helps the software designer, based on his design goals and objectives, to choose the appropriate refactorings that will improve the quality of his design and enables him to predict the quality drift caused by using the refactorings.

ملخص الرسالة

الاســـــــــم: كريم عمر عليش

عنوان الرسالة: تصنيف أساليب إعادة الهيكلية بناء على صفات جودة البرمجيات

التخصـــــص: علوم الحاسب الآلي

تاريخ التخـرج: يونيو ٢٠٠٨

واحدة من اكثر التقنيات المستخدمة لتحسين جودة البرامج تسمى إعادة الهيكلية. إعادة الهيكلية هي عملية تحسين تصميم البرامج عن طريق تغيير هيكلها الداخلي بدون التأثير على السلوك الخارجي. عند تطبيق أساليب إعادة الهيكلية ، بعض صفات الجودة يمكن أن تتحسن والبعض الآخر يمكن أن يضعف. وهذا يعني أن تحسين صفة واحدة قد تؤثر سلبا على نوعية اخرى من الصفات. ومع ذلك ، لا توجد مبادئ توجيهية لمساعدة مصمم البرامج في اختيار بعض أساليب إعادة الهيكلية لتطبيقها من أجل الإستفادة المثلى فيما يتعلق ببعض أهداف التصميم. في هذا البحث، نقترح تصنيف لأساليب إعادة الهيكلية بما فيها إعادة الهيكلية لأنماط بناء على تأثيرهم على صفات جودة البرامج. بالإضافة الى ذلك، لقد تحققنا من صحة هذا التصنيف بإستخدام نظم برامج حقيقية. هذه الدراسة بدورها، تساعد مصمم البرامج، بناء على غاياته وأهدافه، في اختيار أساليب إعادة الهيكلية المناسبة التي من شأنها أن تحسن من جودة التصميم. بالإضافة الى ذلك فإن الدراسة تمكن مصصم البرامج من التنبؤ بجودة البرامج الناجمة عن استخدام أساليب إعادة الهيكلية.

Chapter 1

Introduction

Any software that is related to a real-world problem domain, must continuously adapt to changes as the problem domain changes [18]. During the development of software and its maintenance phase, it is very likely that the code will be modified or improved. This can happen due to many reasons, such as changes in the requirements, the addition of new requirements, or bug fixes. All these changes usually affect the internal structure of software negatively [18]. As a result, this makes new changes difficult to implement and the code drifts away from the original design. In some cases, the changes to software only affect its internal structure; they do not affect its behavior. For instance, a programmer might rename variables or methods, or substitute duplicate code with calls to a method. In object-oriented paradigm, the process of making changes to software that affects its internal structure without altering its behavior is called "refactoring" [29, 44]. Refactoring can be occurred at high-level that is design level and at low-level that is code level such as renaming methods or variables [44].

Refactoring is a technique that reduces software complexity by improving its internal structure. Therefore, refactoring is assumed to positively affect non-functional aspects such as extensibility, modularity, reusability, complexity, maintainability, efficiency [42]. However, refactoring has negative effect, too [42, 51]. In general, improving one quality attribute may affect negatively other quality attributes. Hence, we need to choose the appropriate refactorings to apply in order to improve certain software quality attributes.

Our focus and objective is to propose a classification of refactoring methods including refactoring to patterns based on their measurable effect on software quality attributes in order to help the designer decide which refactoring methods to apply in order to optimize a software system with regard to certain design goals.

In the following sections we state the research problem, the importance of the research problem, the contributions of this thesis, and the organization of the thesis.

1.1. Problem Statement

Refactoring is suggested as a means to improve software quality attributes such as extensibility, modularity, reusability, complexity, maintainability, efficiency [42]. There are many different refactorings each has a particular purpose and effect. When applying refactorings, some quality attributes can be improved and some others can be impaired. This means that improving one quality attribute may affect negatively other quality

attributes. However, there are no guidelines to help the software designer decide which refactorings to apply in order to optimize a software system with regard to certain design goals. Therefore, we need to study the effect of refactoring by classifying different refactorings based on the quality attributes they aim to improve.

1.2. Rationale: Problem Importance

It is expected that refactoring improves the quality of the software system with regard to development activities and future maintenance. However, the effect of refactoring on software quality attributes may vary [42]. For example, some refactorings remove code redundancy, some enhance the reusability, some raise the level of abstraction, some have a negative effect on the performance, and so on.

To perform refactoring on any software project, we need to accomplish certain steps. These steps include: identify the right places to refactor, select the appropriate refactorings to apply, and check if the applied refactorings will improve the overall software quality [18]. These steps become a major task to the software developer. Moreover, the software designers usually design for specific design goals which may contradict with each other. Furthermore, it is unclear for the software designers how to use refactorings in order to improve specific quality attributes.

Our literature survey, discussed in Chapter 3, reveals that only limited number of studies have investigated the effect of refactoring on software quality attributes.

Additionally, it reveals that software engineering community has been suffering from the lack of refactorings classification based on their measurable effect on several internal and external software quality attributes. Consequently, we will look into this problem and classify refactorings according to which software quality attributes they affect. This, in turn, helps the software designer, based on his design goals and objectives, to choose the appropriate refactorings that will improve the quality of his design and enables him to predict the quality drift caused by using the refactorings.

1.3. Research Contributions

The main contributions of this work are the following:

1. Identifying a set of object-oriented metrics that affect external software quality attributes based on available literature.

2. Studying the effect of refactoring methods and refactoring to patterns on internal software quality metrics and external software quality attributes.

3. Proposing a classification of refactoring methods and refactoring to patterns based on their measurable effect on internal and external software quality attributes.

4. Empirically validate the refactoring classification by using real software projects.

1.4. Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides a background on software refactoring, software refactoring to patterns, and software quality attributes. Chapter 3 presents an overview of the related work. Chapter 4 describes the internal and the external software quality attributes and the relationship between them. Chapter 5 classifies refactoring methods and refactoring to patterns based on software quality attributes. Chapter 6 empirically validates the refactoring classification by using real software projects. Chapter 7 concludes the thesis with summary of main contributions and directions for future work.

Chapter 2

Background

In this chapter, we provide a brief overview on software refactoring, software refactoring to patterns, and software quality attributes.

2.1. Software Refactoring

This section defines the concept of refactoring, presents the benefits of refactoring, describes refactoring catalog, and explains the refactoring methods used in this thesis.

2.1.1. Definition of Refactoring

The term "refactoring" was first introduced by Opdyke in his PhD dissertation [44]. Refactoring refers to "the process of changing an [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [29]. In other words, refactoring improves the internal structure of software without adding new functionality. If we look at the definition we can observe the following: (*i*) refactoring improves the maintenance of a software project (*ii*)

refactoring helps us to rebuild the source code in order to make it easier to understand and modify it (*iii*) refactoring preserves the behavior of software systems. This means that by applying refactorings, for example, unused parameters or variables can be removed, duplicated code can be removed, and unbalanced responsibilities between entities can be redistributed. Fowler defines in his book "refactoring (noun)" and "refactor (verb)" as follows [29]:

- **Refactoring** (noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing the observable behavior of the software.

- **Refactor** (verb): To restructure software by applying a series of refactorings without changing the observable behavior of the software.

2.1.2. Benefits of Refactoring

Refactoring is a tool that can be used for several purposes [29]:

- **Improves the design of software**: during accumulating code changes, code loses its structure and drifts away from its original design. Refactoring can be used to improve the design of software by redistributing parts of the code to the "right" places, and by removing duplicated code.

- **Makes software easier to understand**: refactoring can help to make the code more readable.

- **Helps to find bugs**: by clarifying the structure of the code, certain assumptions within the code are also clarified, making it easier to find bugs.

- **Helps to program faster**: through improving the design and overall understandability of the code, refactoring supports rapid software development.

2.1.3. Refactoring Catalog

Fowler [29] defines more than 70 different kinds of code refactorings over 6 categories in his refactoring catalog. Each one of them includes the motivation of why the refactoring should be performed and step-by-step description of how to carry out the refactoring. After that many other refactorings have been discovered and used. Some useful examples of refactorings include the following:

- **Composing Methods**: extract method, inline method, and inline temp.

- **Simplifying Conditional Expressions**: decompose conditional, introduce null object, and consolidate conditional expression.

- **Moving Features between Objects**: extract class, inline class, move method, and move filed.

- **Organizing Data**: encapsulate field, replace magic number with symbolic constant, and replace data value with object.

- **Dealing with Generalization**: extract interface, extract subclass, pull up method, and replace inheritance with delegation.

- **Making Method Calls Simpler**: remove setting method, hide method, rename method, and remove parameter.

2.1.4. Refactoring Methods under Study

This section briefly explains with the examples the refactoring methods used in this thesis which include the following:

- **Consolidate Conditional Expression**: if we have a sequence of conditional tests with the same result, then we can combine them into a single conditional expression and extract it into new method. As explained in [29], Figure 2-1 shows an example of a code fragment before and after the application of “Consolidate Conditional Expression”.

```
// Before
double disabilityAmount() {

  if (seniority < 2) return 0;
  if (monthsDisabled > 12) return 0;
  if (isPartTime) return 0;

  // compute the disability amount
  ……
}

// After
double disabilityAmount() {

  if (isNotEligableForDisability())
      return 0;
  // compute the disability amount
  ……
}

boolean isEligibleForDisability() {
  return ((seniority < 2) || (monthsDisabled > 12)
          || (isPartTime));
}
```

Figure 2-1. Consolidate conditional expression: before and after

- **Encapsulate Field**: for each public filed (attribute) in a class, encapsulate this filed by making it private and providing methods for accessing (getting) and updating (setting) its value [29]. In addition, replace all direct references to the attribute by calls to these methods. Figure 2-2 shows an example of a code fragment before and after the application of "Encapsulate Field".

```
// Before
class customer{
  public String name
  ……

}

// After
class customer{
  private String name;
  ……
  public String getName()
     {return name;}
  public void setName(String arg)
     {name = arg;}
}
```

Figure 2-2. Encapsulate field: before and after

- **Extract Class**: when we have a class doing work that should be done by two, then we need to extract a new class from the original class and move the relevant fields and methods from the old class into the new class [29]. As a result, each class will have clear responsibilities. Figure 2-3 shows an example of UML class diagram before and after the application of "Extract Class". This example is cited from Fowler's book [29].

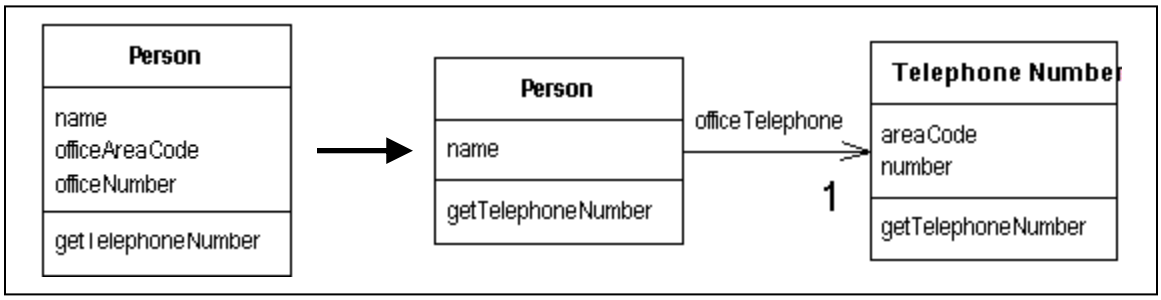

Figure 2-3. Extract class: before and after

- **Extract Method**: this technique extracts a set of statements that can be grouped together into a new method whose name explains the purpose of the method [29]. Then, it replaces the extracted statements with a call to a new method. "Extract Method" allows the extracted method to be reused in other places, and makes the code more readable. Figure 2-4 shows an example of a code fragment before and after the application of "Extract Method".

```
// Before
void printOwing(double amount){

   printBanner();

   //print details
   System.out.println ("name:" + name);
   System.out.println ("amount" + amount);
}

// After
void printOwing(double amount){
   printBanner();
   printDetails(amount);
}

void printDetails(double amount){
   System.out.println ("name:" + name);
   System.out.println ("amount" + amount);
}
```

Figure 2-4. Extract method: before and after

- **Hide Method**: if a method is not used by any other classes, then change its visibility by making the method private. Figure 2-5 shows an example of UML class diagram before and after the application of "Hide Method", where '+' means public and '-' means private. This example is cited from Fowler's book [29].

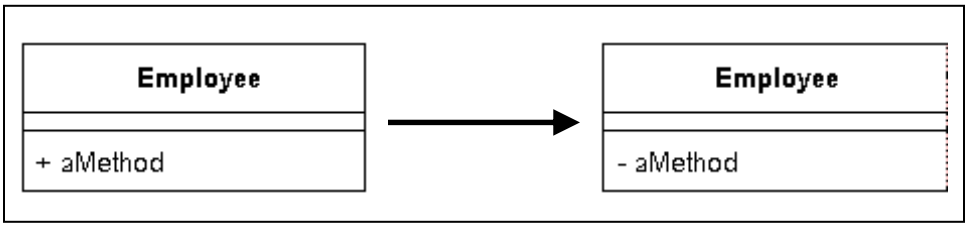

Figure 2-5. Hide method: before and after

- **Inline Class**: we have a class which is not doing very much, hence move all its attributes and methods into another class and delete it. “Inline Class” is the reverse of “Extract Class”. Figure 2-6 shows an example of UML class diagram before and after the application of “Inline Class”. This example is cited from Fowler’s book [29].

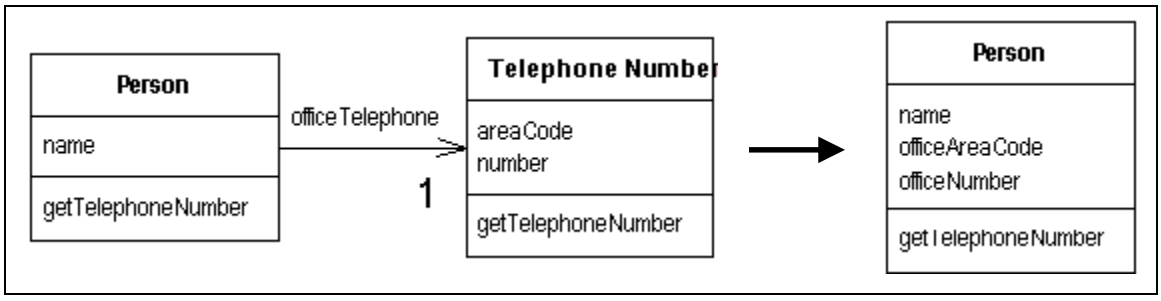

Figure 2-6. Inline class: before and after

- **Inline Method**: this technique allows you to place a method's body into the body of its callers and remove the method [29]. It can be used when a method's body is as clear as its name. An example of a code fragment before and after the application of "Inline Method" is shown in Figure 2-7.

```
// Before
int getRating() {
   return (moreThanFiveLate()) ? 2:1;
}
boolean moreThanFiveLate() {
   return numberOfLate > 5;
}

// After
int getRating() {
   return (numberOfLate > 5) ? 2:1;
}
```

Figure 2-7. Inline method: before and after

- **Inline Temp**: when we have a temp that is assigned to once with a simple expression and it is getting in the way of other refactoring, then replace all references to that temp with the expression [29]. Figure 2-8 shows an example of a code fragment before and after the application of "Inline Temp".

```
// Before
boolean ckeckPrice(){

   double basePrice = anOrder.basePrice();
   return (basePrice > 1000);
}

// After
boolean ckeckPrice(){

   return (anOrder.basePrice() > 1000);
}
```

Figure 2-8. Inline temp: before and after

- **Remove Setting Method**: for any field that should be set at creation time and never altered, remove any setting method for that field [29]. This means that if you do not want a field to change once the object is created, then do not provide a setting method for that field. An example of a code fragment before and after the application of "Remove Setting Method" is shown in Figure 2-9.

```
// Before
class Account {
   private String id;
   private String name;
   public Account (String arg1, String arg2) {
      id = arg1;
      name = arg2;
   }

   public void setId (String arg) {
      id = arg;
   }
}

// After
class Account {
   private String id;
   private String name;
   public Account (String arg1, String arg2) {
      id = arg1;
      name = arg2;
   }
}
```

Figure 2-9. Remove setting method: before and after

- **Replace Assignment with Initialization**: this technique is introduced in the refactoring website [10]. In general, the programmers first declare lots of variables that will be used in the program, and then, assign values to them. Instead of that make direct internalizations to these variables. Figure 2-10 shows an example of a code fragment before and after the application of "Replace Assignment with Initialization".

```
// Before
void foo() {
   int i;
   // ....
   i = 7;
}

// After
void foo() {
   // ...
   int i = 7;
}
```

Figure 2-10. Replace assignment with initialization: before and after

- **Replace Magic Number with Symbolic Constant**: sometimes you have numbers (magic numbers) with special values that are difficult to understand their meaning. To solve this problem, create a constant with useful name and replace the number with it [29]. When you use a number in many places and you want to change it, this technique will help you to change the number in one place only. Figure 2-11 shows an example of a code fragment before and after the application of "Replace Magic Number with Symbolic Constant".

```
// Before
double potentialEnergy(double mass, double height){

   return mass * 9.81 * height;
}

// After
static final double GRAVITATIONA_ CONSTANT = 9.81;

double potentialEnergy(double mass, double height){
   return mass * GRAVITATIONAL_CONSTANT * height;
}
```

Figure 2-11. Replace magic number with symbolic constant: before and after

- **Reverse Conditional**: this is technique is introduced in the refactoring website [10]. It allows you to reverse the sense of the conditional and reorder the conditional's clauses [10]. This will make the conditional easier to understand. Figure 2-12 shows an example of a code fragment before and after the application of "Reverse Conditional".

```
// Before
if ( !isSummer( date ) )
  charge = winterCharge(quantity);

else
  charge = summerCharge(quantity);


// After
if ( isSummer( date ) )
  charge = summerCharge(quantity);

else
  charge = winterCharge(quantity);
```

Figure 2-12. Reverse conditional: before and after

2.2. Software Refactoring to Patterns

This section defines the concept of refactoring to patterns, describes refactoring to patterns catalog, and explains the refactoring to patterns methods used in this thesis.

2.2.1. Definition of Refactoring to Patterns

The concept of refactoring to patterns is introduced in "Refactoring to Patterns" book by Joshua Kerievsky [39]. Refactoring to patterns explains the relation between refactoring and design patterns. More precisely, Kerievsky defines refactoring to patterns as

"marriage of refactoring (the process of improving the design of existing code) with patterns (the classic solutions to recurring design problems)" [39].

2.2.2. Refactoring to Patterns Catalog

Kerievsky [39] defined 27 pattern-directed refactorings over 6 categories in his book. Each one of them includes the motivation of why the refactoring should be performed and step-by-step description of how to carry out the refactoring. Some useful examples of refactoring to patterns include the following:

- **Creation**: inline singleton, and replace constructors with creation methods.
- **Simplification**: compose method, and replace conditional dispatcher with command.
- **Generalization**: form template method, and extract composite.
- **Protection**: introduce null object, and replace type code with class.
- **Accumulation**: move accumulation to collecting parameter.
- **Utilities**: chain constructors, and unify interfaces.

2.2.3. Refactoring to Patterns under Study

In this section, we briefly explain with the examples the refactoring to patterns methods used in this thesis.

- **Chain Constructors**: when we have multiple constructors that contain duplicate code, then we chain the constructors together to obtain the least amount of duplicate code [39]. Figure 2-13 shows an example of a code fragment before and after the application of “Chain Constructors”. This example is cited from Kerievsky’s book [39].

```
// Before
public class Loan {
    ……
    public Loan(float notional, float outstanding, int rating, Date expiry) {
       this.strategy = new TermROC();
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
    }
    public Loan(float notional, float outstanding, int rating, Date expiry,
                Date maturity) {
       this.strategy = new RevolvingTermROC();
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }
    public Loan(CapitalStrategy strategy, float notional, float outstanding,
                int rating, Date expiry, Date maturity) {
       this.strategy = strategy;
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }

}

// After
public class Loan {
    ……
    public Loan(float notional, float outstanding, int rating, Date expiry) {
        this(new TermROC(), notional, outstanding, rating, expiry, null);
    }
    public Loan(float notional, float outstanding, int rating, Date expiry,
                Date maturity) {
        this(new RevolvingTermROC(), notional, outstanding, rating, expiry,
             maturity);
    }
    public Loan(CapitalStrategy strategy, float notional, float outstanding,
                int rating, Date expiry, Date maturity) {
       this.strategy = strategy;
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }
}
```

Figure 2-13. Chain constructors: before and after

- **Compose Method**: if a method's logic can not be rapidly understood, then convert the logic into a small number of intention-revealing steps at the same level of detail [39]. Figure 2-14 shows an example of a code fragment before and after the application of "Compose Method". This example is cited from Kerievsky's book [39].

```
// Before
public void add(Object element) {
   if (!readOnly) {
      int newSize = size + 1;
      if (newSize > elements.length) {
         Object[] newElements = new Object[elements.length + 10];
          for (int i = 0; i < size; i++)
             newElements[i] = elements[i];
          elements = newElements;
      }
      elements[size++] = element;
   }
}

// After
public void add(Object element) {
   if (readOnly)
      return;
   if (atCapacity())
      grow();
   addElement(element);
}
private boolean atCapacity() {
   return (size + 1) > elements.length;
}
private void grow() {
   Object[] newElements = new Object[elements.length + 10];

   for (int i = 0; i < size; i++)
      newElements[i] = elements[i];
   elements = newElements;
}
private void addElement(Object element) {
   elements[size++] = element;
}
```

Figure 2-14. Compose method: before and after

- **Form Template Method**: this technique is typically used if there are two methods in subclasses perform similar steps in the same order, however the steps are different. It generalizes the methods by extracting their steps into methods with identical signatures, then pull up the generalized methods to form a Template Method [39]. Figure 2-15 shows an example of UML class diagram before and after the application of "Form Template Method". This example is cited from Kerievsky's book [39].

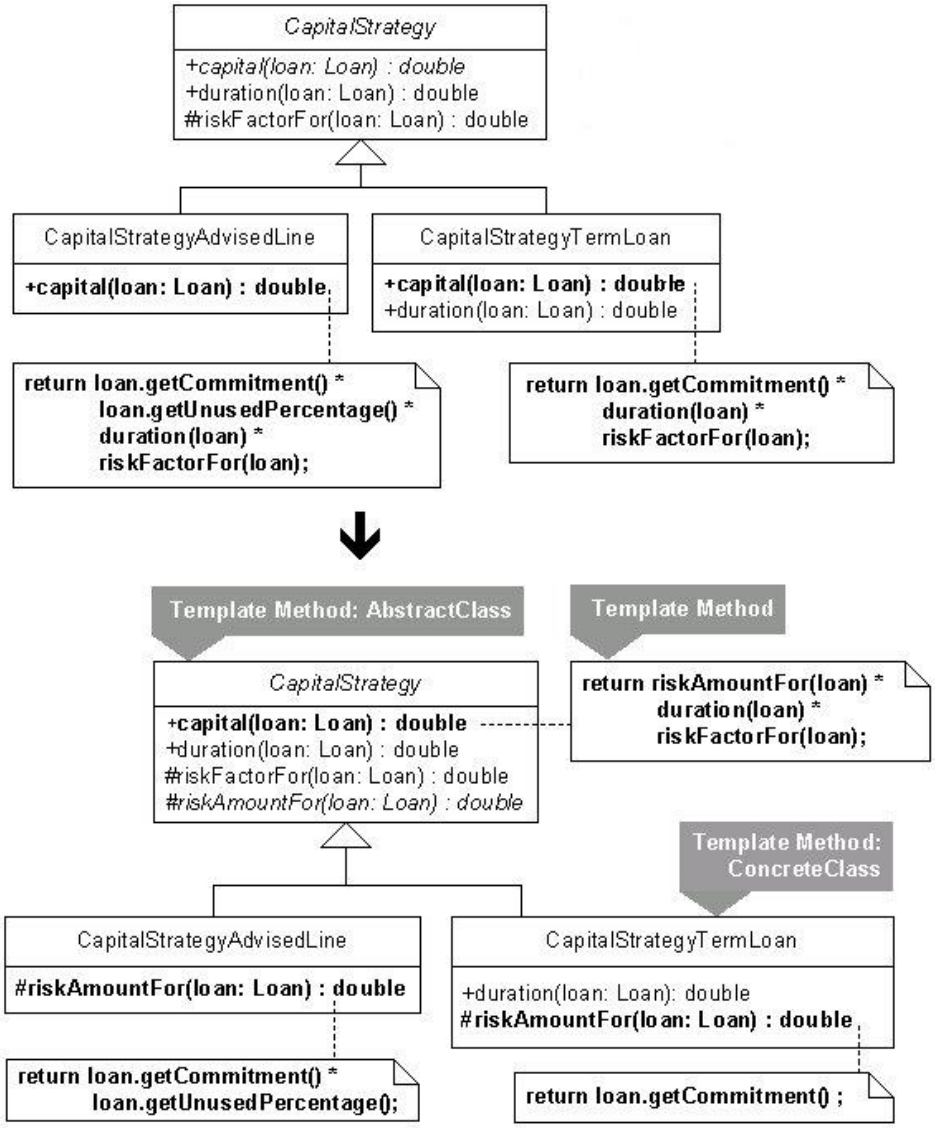

Figure 2-15. Form template method: before and after

- **Introduce Null Object**: this technique allows you to deal with repeated checks for a null field or variable by replacing the "null value" with a "null object" [39]. This object provides the appropriate null behavior. Figure 2-16 shows an example of UML class diagram before and after the application of "Introduce Null Object". This example is cited from Kerievsky's book [39].

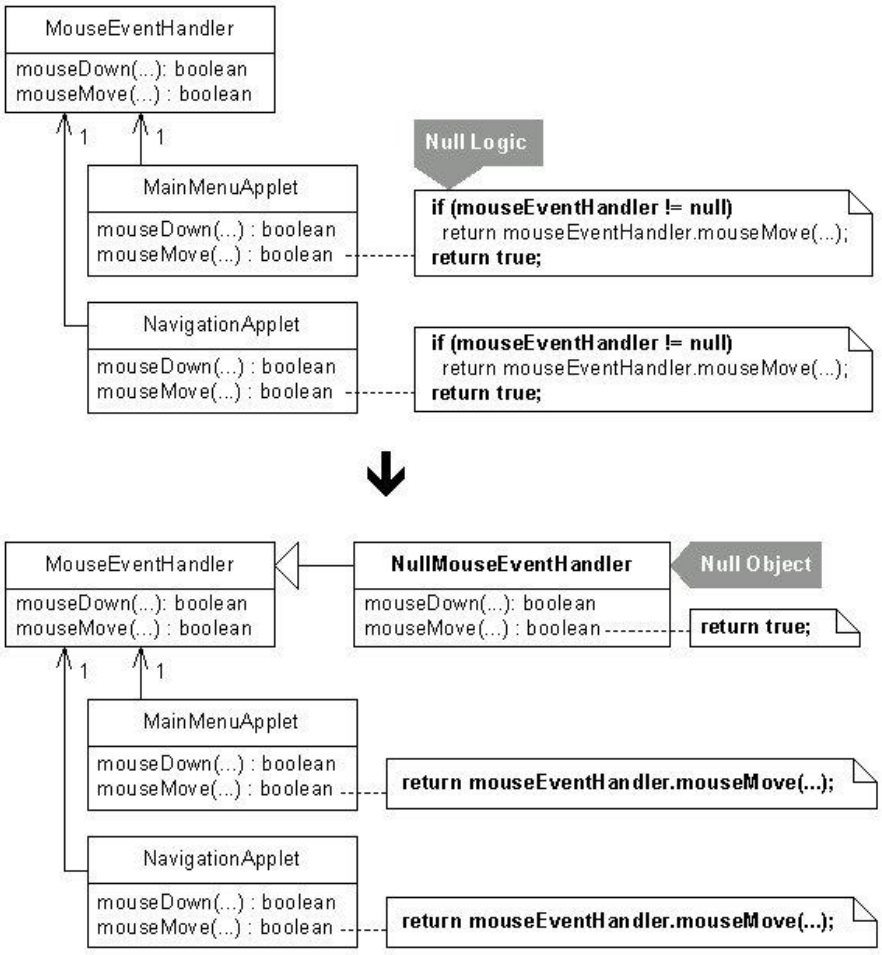

Figure 2-16. Introduce null object: before and after

- **Replace Conditional Dispatcher with Command**: if you use conditional logic to dispatch requests and execute actions, then create a command for each action, store the commands in a collection, and replace the conditional logic with code to fetch and execute commands [39]. Figure 2-17 shows an example of UML class diagram before and after the application of "Replace Conditional Dispatcher with Command". This example is cited from Kerievsky's book [39].

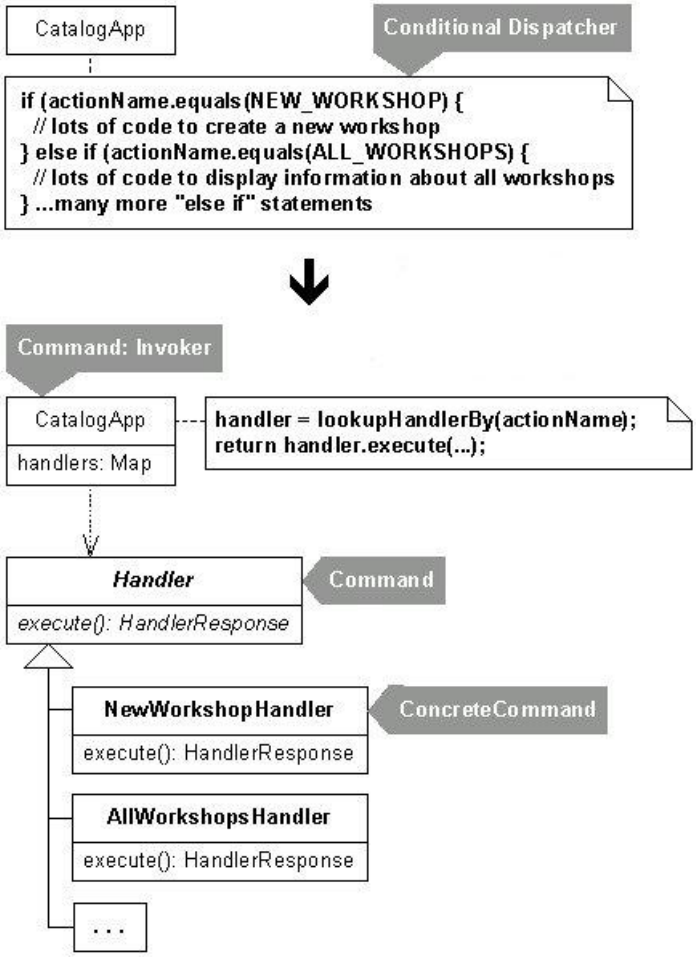

Figure 2-17. Replace conditional dispatcher with command: before and after

- **Replace Constructors with Creation Methods**: it is hard to decide which constructor of a given class (class with many constructors) to call during development. In this case, we can replace the constructors with intention-revealing creation methods that return object instances [39]. Figure 2-18 shows an example of UML class diagram before and after the application of "Replace Constructors with Creation Methods". This example is cited from Kerievsky's book [39].

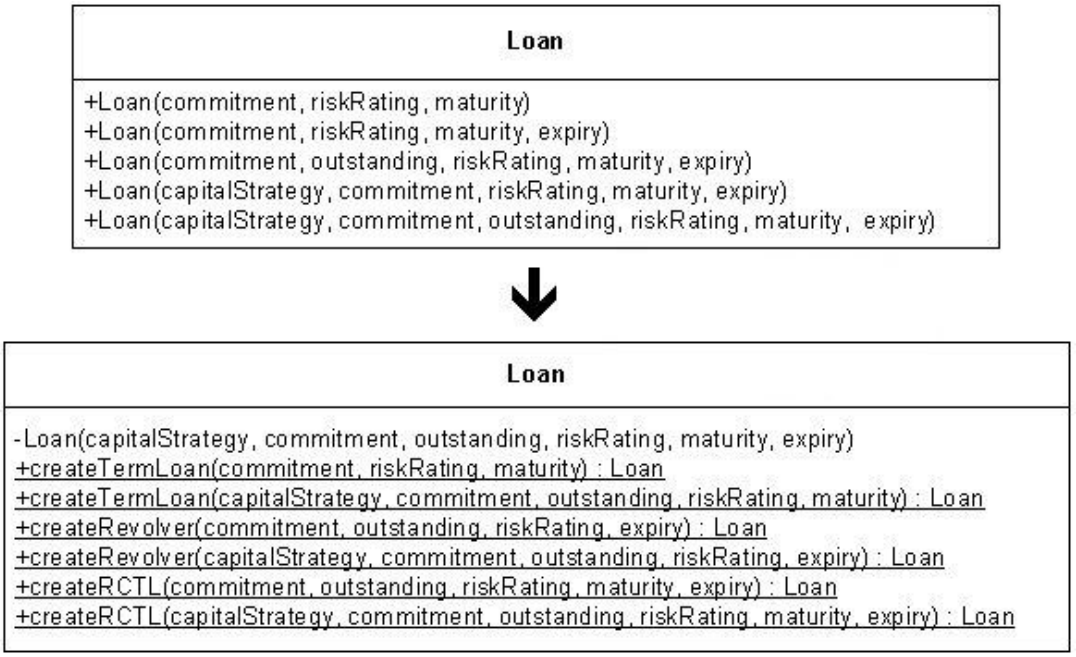

Figure 2-18. Replace constructors with creation methods: before and after

- **Unify Interfaces**: this technique allows you to provide interface to a superclass that is identical to subclass's interface. To do so, find all public methods on the subclass that are missing on the superclass. Then, add copies of these missing methods to the superclass and changing each one to perform null behavior [39]. Figure 2-19 shows an example of UML class diagram before and after the application of "Unify Interfaces". This example is cited from Kerievsky's book [39].

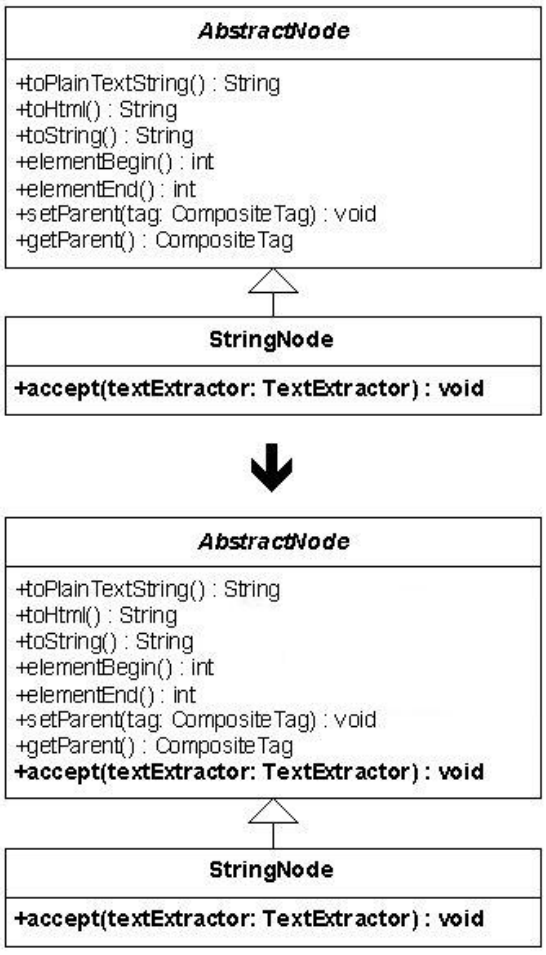

Figure 2-19. Unify interfaces: before and after

2.3. Software Quality Attributes

Software quality is the degree to which software possesses a desired combination of attributes (e.g., adaptability, reusability) [35]. This means that defining software quality for a system is equivalent to define a list of software quality attributes for that system [35]. ISO/EIC 9126 standard [37] defines software quality characteristics as "a set of attributes of a software product by which its quality is described and evaluated". The factors that affect software quality can be classified into two groups [45]: (*i*) factors that can be directly measured i.e. internal quality attributes (e.g. length of program as lines of code) and (*ii*) factors that can be measured only indirectly i.e. external quality attributes (e.g. maintainability and reliability). In the following subsections, we provide a brief overview on internal and external software quality attributes.

2.3.1. Internal Quality Attributes

Internal attributes of software product, process, or resource are those which can be defined (measured) purely in terms of software product, process, or resource itself [27]. Software metrics are typically used as internal quality attributes [28]. A software metric is a characteristic (attribute) of software product, process, or resource [28, 45]. Software metrics play an important role in planning, managing and controlling software development projects. They can be used to extract useful and measurable information about the structure of a software system [47]. Nowadays, there are lots of metrics suites available used to understand the structure of software system and evaluate its quality.

Some popular metrics suites for object-oriented systems are proposed by Chidamber & Kemerer [25], Henry and Kafura [33, 34], and Briand et al. [21, 22].

2.3.2. External Quality Attributes

External attributes of software product, process, or resource are those which can only be defined (measured) with respect to how the software product, process, or resource relates to other entities in its environment [27]. The ISO/EIC 9126 standard identifies six quality attributes for software product [37]:

- **Functionality**: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

- **Reliability**: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- **Usability**: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

- **Efficiency**: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

- **Maintainability**: A set of attributes that bear on the effort needed to make specified modifications.
- **Portability**: A set of attributes that bear on the ability of software to be transferred from one environment to another.

Measurements of external attributes involve measurement of one or more internal attributes [27]. For example, "reliability" of software is dependent on many factors, such as the program itself, the compiler, the machine, and the user. Several empirical studies attempt to assess external quality attributes by using internal quality attributes such as [14, 23, 24, 26, 41, 50].

Chapter 3

Literature Review

Software refactoring has been studied extensively by the software engineering research community. However, only limited number of studies have investigated the effect of refactoring on software quality attributes. In this chapter, we provide an overview of the previous studies that have investigated the effect of refactoring on internal and external software quality attributes.

3.1. Assessing Refactoring Effect on Internal Software Quality

Tahvildari and Kontogiannis [49] proposed a framework in which a catalog of object-oriented metrics can be used as an indicator to automatically detect where a particular transformation can be applied to improve the software quality. This is achieved by analyzing the impact of various meta-pattern transformations (abstraction, extension, movement, encapsulation, buildrelation, and wrapper) on these object-oriented metrics. The selected object-oriented metrics were complexity (CDE, NOM, WMC), coupling (DAC, RFC), and cohesion (LCOM, TCC). They consider their approach to be helpful to

a designer or programmer by suggesting proper meta-pattern transformations and useful to develop and maintain good quality software.

Stroggylos and Spinellis [47] analyzed source code version control system logs of four popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by refactorings. The metrics examined include C & K metrics [25], Ca (Afferent Coupling), and NPM (Number of Public Methods). Their results indicate a significant change of certain metrics to the worse. Specifically refactorings caused an increase in metrics such as LCOM, Ca, and RFC indicating that refactorings caused classes to become less coherent.

Bois and Mens [17] propose formalism based on abstract syntax tree representation of the source-code, extended with cross-references to describe the impact of refactoring on internal program quality. They focused on three refactoring methods: “Encapsulate Filed”, “Pull up Method”, and “Extract Method” to analyze their impact on internal program quality metrics i.e. NOM, NOC, CBO, RFC, and LCOM. They found that “Encapsulate Filed” and “Extract Method” increase NOM, RFC, and LCOM. In addition, the effect of “Pull up Method” on superclass increases NOM, RFC, LCOM, and CBO. However, the effect of “Pull up Method” on subclass decreases NOM, RFC, LCOM, and CBO.

3.2. Assessing Refactoring Effect on External Software Quality

Tahvildari and Kontogiannis [48] encoded design decisions as soft-goal graphs to examine the dependencies between them and guide the application of the transformation process. They studied the association of design pattern transformations (abstraction, extension, movement, encapsulation, buildrelation, and wrapper) with a possible effect on soft-goals to address maintainability enhancement.

Bois et al. [15] performed a controlled experiment to empirically investigate differences in program comprehension between the application of Refactor to Understand and the traditional Read to Understand pattern. Refactor to Understand is a reverse engineering pattern that is used to improve the code and the maintainers understanding. Their findings provide the first empirical support for the claim that refactoring can be used to improve the understanding of software.

Geppert et al. [30] empirically investigated the impact of refactoring of a legacy system on changeability. They considered three factors for changeability: customer reported defect rates, effort, and scope of changes. They found a significant decrease in customer reported defect rates and in effort needed to make changes in the post-refactoring releases.

Wilking et al. [51] investigated the effect of refactoring on maintainability and modifiability through an empirical evaluation. Maintainability was tested by randomly

inserting defects into the code and measuring the time needed to fix them. Modifiability was tested by adding new requirements and measuring the time and LOC metric needed to implement them. The direct effect of an increased maintainability and a better modifiability caused by refactoring could not be shown within this controlled experiment.

3.3. Mapping Refactoring Effect on Internal Software Quality to External Software Quality

Kataoka et al. [38] proposed coupling metrics as a quantitative evaluation method to measure the effect of refactoring on the maintainability of the program by comparing the coupling before and after the refactoring. They focused on refactorings that are supposed to reduce coupling among methods such as “Extract Method” and “Extract Class”. They concluded that this approach was really effective to quantify the refactoring effect and helped the developers and analysts to make reasonable decision on how to improve their products’ maintainability.

Bois et al. [16] developed practical guidelines for applying refactoring methods to improve coupling and cohesion characteristics and validated these guidelines on an open source software system. They assumed that coupling and cohesion are quality attributes which are generally recognized as indicators for software maintainability. The set of refactoring methods used in this study was “Extract Method”, “Move Method”, “Replace Method with Method Object”, “Replace Data Value with Object”, and “Extract Class”. The results indicate that these guidelines can be used to improve coupling and cohesion

and therefore led to improving maintainability except for “Extract Method” which does not help in improving neither coupling nor cohesion.

Moser et al. [43] proposed a methodology to assess if refactoring improves reusability and promotes ad-hoc reuse in an XP-like development environment. They focused on internal software metrics (CBO, LCOM, RFC, WMC, DIT, NOC, LOC, and McCabe’s cyclomatic complexity) that are considered to be relevant for reusability. A case study was conducted on software project developed using an agile, XP-like methodology to analyze the impact of refactoring on internal quality metrics. Their results indicate that refactoring has a positive effect on reusability and therefore it promotes ad-hoc reuse in an XP-like development environment.

A summary of the related work is presented in Table 3-1. Based on the overview of the related work and Table 3-1, we can observe the limitations of the existing research studies reviewed above. The researchers in [48, 49] have focused on transformations at the level of design patterns, they did not consider lower level refactorings proposed by Fowler [29]. Furthermore, [15, 30, 43, 47, 51] did not investigate the direct effect of each kind of refactoring methods on internal or external quality attributes. In addition, [17, 47, 49] did not relate the effect of refactoring on internal quality metrics to the external quality attributes. Moreover, [16, 17] have classified refactoring methods based on internal quality metrics using limited number of refactoring methods. Additionally, none

of the existing research studies has classified refactoring methods and refactoring to patterns methods based on their effect on several external software quality attributes.

Accordingly, the originality of this research focuses on classifying refactoring methods and refactoring to patterns methods based on their measurable effect on several software quality attributes including internal and external quality in order to improve specific quality attributes that are indicators for a good software quality.

Table 3-1. Summary of related work

Study	Internal Quality	External Quality	Refactoring Level	Refactoring Classification?
Tahvildari and Kontogiannis [49]	RFC, NOM, CDE, DAC, TCC, LCOM, WMC	-	Design level	Yes, based on internal quality. Limited to 6 meta-pattern transformations.
Stroggylos and Spinellis [47]	C & K metrics, Ca, NPM	-	Code level	No
Bois and Mens [17]	NOM, NOC, CBO, RFC, LCOM	-	Code level	Yes, based on internal quality. Limited to 3 refactoring methods.
Tahvildari and Kontogiannis [48]	-	Maintainability	Design level	No
Bois et al. [15]	-	Understandability	Code level	No
Geppert et al. [30]	-	Changeability	Code level	No
Wilking et al. [51]	-	Maintainability Modifiability	Code level	No
Kataoka et al. [38]	Coupling	Maintainability	Code level	No
Bois et al. [16]	Coupling, Cohesion	Maintainability	Code level	Yes, based on internal quality. Limited to 5 refactoring methods.
Moser et al. [43]	C & K metrics, MCC, LOC	Reusability	Code level	No
Our Work	C & K metrics, FOUT, NOM, LOC	Adaptability Completeness Maintainability Understandability Reusability Testability Reliability	Code level	Yes, based on internal & external quality. Consider 12 refactoring methods & 7 refactoring to patterns.

Chapter 4

Relating Internal Software Quality to External Software Quality

Quality is an important factor for the success of any software product. Software developers aim to achieve high-quality software [45]. Pressman defined software quality as "conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software" [45]. As it is explained earlier in Chapter 2, there are two factors that affect software quality: internal quality attributes and external quality attributes.

This chapter defines the internal and the external software quality attributes which we used in refactoring classification and describes how external quality attributes can be assessed using internal quality attributes.

4.1. Internal Quality Attributes under Study

Software metrics are typically used as internal quality attributes [28]. A software metric is a characteristic (attribute) of software product, process, or resource [28, 45]. In order to classify the refactorings based on internal quality attributes, we used a suite of metrics that measures the structural quality of object-oriented code and design. More specifically, we consider the C&K metrics suite [25] which consists of Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response for a Class (RFC), Weighted Methods per Class (WMC), and Lack of Cohesion on Methods (LCOM). In addition, we include one metric which measures class coupling (Fan out) and two metrics which measure class size (Number of Methods and Lines of Code). We chose these metrics because of their wide acceptance among the software engineering community. Additionally, they have been used by several previous empirical studies such as [14, 23, 24, 26, 41, 50] to investigate their correlation with external software quality attributes. Moreover, they capture important dimensions of object-oriented design characteristics: inheritance by (DIT, NOC), coupling by (CBO, RFC, FOUT), size/complexity by (WMC, NOM, LOC), and cohesion by (LCOM). The metrics we investigated are the following:

- **Depth of Inheritance Tree (DIT)**: it is defined as the length of the longest path from a given class to the root class in the inheritance hierarchy.

- **Number of Children (NOC)**: it is defined as the number of classes that inherit directly from a given class.

- **Coupling between Objects (CBO)**: it is defined as the number of distinct noninheritance related classes to which a given class is coupled. A class is coupled to another if it uses methods or attributes of the coupled class.

- **Response for a Class (RFC)**: it is defined as the number of methods that can potentially be executed in response to a message being received by an object of that class.

- **Fan out (FOUT)**: it is defined as the number of classes that a given class uses, not the classes it is used by.

- **Weighted Methods per Class (WMC)**: it is defined as the number of methods defined (implemented) in a given class. Traditionally, it measures the complexity of an individual class (weighted sum of all the methods in a class). However, based on [25], if we consider all methods of a class equally complex, then WMC is simply the number of methods defined in each class. In this thesis, we consider all methods of a class to be equally complex.

- **Number of Methods (NOM)**: it is defined as the number of methods implemented in a given class.

- **Lines of Code (LOC)**: it is defined as the total source lines of code in a class excluding all blank and comment lines.

- **Lack of Cohesion on Methods (LCOM)**: it is defined as the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever this subtraction is negative.

4.2. External Quality Attributes under Study

To classify the refactorings according to external quality attributes, we identified a set of external quality attributes as follows:

- **Adaptability**: it is defined as the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [36].

- **Completeness**: it is defined as the degree to which the component implements all required capabilities [46].

- **Maintainability**: it is defined as the ease with which a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [36].

- **Understandability**: it is defined as the degree to which the meaning of a software component is clear to a user [46].

- **Reusability**: it is defined as the degree to which a component can be used in more than one software system, or in building other components, with little or no adaptation [46].

- **Testability**: it is defined as a set of attributes of software that bear on the effort needed to validate the software product [37].

- **Reliability**: it is defined as the ability of a component to perform its required functions under stated conditions for a specified period of time [36].

4.3. External Quality Attributes Assessments Using Internal Quality Attributes

Software metrics can be used as indicators (estimators/predictors) for external software quality attributes. These indicators help software manager to estimate and control the software more effectively. Several empirical studies such as [14, 23, 24, 26, 31, 41, 50] used software metrics as indicators to assess external quality attributes. These empirical studies provide some empirical evidence indicating that most of object-oriented metrics can be useful quality indicators.

The following subsections demonstrate based on available research studies how the internal quality attributes (described in Section 4.1) can be used as indicators to external quality attributes (described in Section 4.2).

4.3.1. Adaptability, Completeness, Maintainability, Understandability, and Reusability Assessments Using Internal Quality Metrics

Dandashi [26] demonstrated a method for assessing indirect quality attributes (adaptability, completeness, maintainability, understandability, and reusability) of object-oriented systems from the direct quality attributes (McCabe's Cyclomatic Number, Halstead's Volume, WMC, DIT, NOC, RFC, CBO, and LOC). A case study of the feasibility of applying direct measurements to assess the indirect quality attributes was conducted using C++ code components taken from the object-oriented Particle-In-Cell Simulations (PICS) problem domain. A survey was conducted to gather data about the PICS indirect quality attributes. The results show that indirect quality attributes measured by human analysis with direct quality attributes measured by the automated tool provide empirical evidence that direct and indirect quality attributes do correlate. More specifically, Complexity, Volume, WMC, and LOC are proportional (positively correlated) to the levels of adaptability, completeness, maintainability, and understandability while NOC, DIT, RFC, and CBO are inversely proportional (negatively correlated) to the levels of adaptability, completeness, maintainability, and understandability. In addition, the reusability can be estimated from adaptability, completeness, maintainability, and understandability.

4.3.2. Testability Assessment Using Internal Quality Metrics

Bruntink and Deursen [24] identified and evaluated a set of object-oriented metrics that can be used to estimate the testing effort of the classes of object-oriented software. LOCC

(Lines of Code for Class) and NOTC (Number of Test Cases) are used to represent the size of a test suite, while DIT, FOUT, LCOM, LOC, NOC (Number of Children), NOF (Number of Fields), NOM (Number of Methods), RFC, and WMC are used as predictors to predict/estimate the size of a test suite. In their experiment, they used five case studies of Java systems for which JUnit test cases exist. The systems include four commercial systems and one open source system (Apache Ant). They were able to find a significant correlation between object-oriented metrics (RFC, FOUT, WMC, NOM, and LOC) and the size of a test suite (LOCC and NOTC). This means that RFC, FOUT, WMC, NOM, and LOC are indicators for testing effort.

4.3.3. Reliability Assessment Using Internal Quality Metrics

Basili et al. [14] empirically investigated the impact of object-oriented design metrics suite introduced by Chidamber and Kemerer [25] (DIT, WMC, RFC, NOC, LCOM, and CBO) on the prediction of fault-prone classes. Their goal is to assess these metrics as predictors of faulty classes and determine whether they can be used as early quality indicators. The data was collected from the development of eight medium-sized information management systems. From their results, five out of the six C&K metrics appear to be useful to predict class fault-proneness. More specifically, they found a significant positive correlation between DIT, WMC, RFC, NOC, and CBO metrics and the probability of detecting faulty classes.

Tang et al. [50] investigated the correlation between object-oriented design metrics (DIT, WMC, RFC, NOC, and CBO) and the probability of the occurrence of object-oriented faults. Their empirical study was conducted on three industrial real-time systems which are subsystems of HMI (Human Machine Interface) software. They extracted the faults information by analyzing the trouble reports of the three systems recorded for the past three years. Their validation suggests that WMC and RFC are significant indicators for fault-prone classes.

Gyimo´thy et al. [31] empirically validated the correlation between a set of object-oriented metrics (including all C&K metrics [25] and LOC metric) and fault-proneness of classes. The experiment was conducted on large open source system called Mozilla and the faults (bugs) information was extracted from its bug database called Bugzilla. They found that LOC and all the C&K metrics were significant except NOC. This means that DIT, CBO, RFC, WMC, LCOM, and LOC are predictors for fault-prone classes.

Reliability is considered to be inversely proportional to the number of faults. For example, when the number of faulty classes increases, the system will be less reliable and vice versa. In the research studies reviewed above, Basili et al. [14], Tang et al. [50], and Gyimo´thy et al. [31] focused on the correlation between set of object-oriented design metrics and fault-prone classes. They found that DIT, NOC, CBO, RFC, WMC, LOC, and LCOM metrics are positively correlated to fault-proneness. This means that DIT,

NOC, CBO, RFC, WMC, LOC, and LCOM metrics are negatively correlated to reliability. Therefore, we consider all object-oriented metrics that are positively correlated to fault-proneness to be negatively correlated to the levels of reliability.

Table 4-2 summarizes the relationship between the internal quality attributes (DIT, NOC, CBO, RFC, FOUT, WMC, NOM, LOC, and LCOM) and the external quality attributes (adaptability, completeness, maintainability, understandability, reusability, testability, and reliability), where "+ve" sign represents positive correlation, "-ve" sign represents negative correlation, "0" sign represents no correlation at all, and "*NA*" sign represents that the correlation between an internal quality attribute and an external quality attribute was not studied.

Table 4-2. The relationship between internal and external software quality attributes

External Quality	Study	Internal Quality								
		Inheritance		Coupling			Size			Cohesion
		DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Adaptability	Dandashi (2002) [26]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Completeness	Dandashi (2002) [26]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Maintainability	Dandashi (2002) [26]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Understandability	Dandashi (2002) [26]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Reusability	Dandashi (2002) [26]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Testability	Bruntink and Deursen (2006) [24]	0	0	*NA*	+ve	+ve	+ve	+ve	+ve	0
Reliability	Basili et al. (1996) [14]	-ve	-ve	-ve	-ve	*NA*	-ve	*NA*	*NA*	0
	Tang et al. (1999) [50]	0	0	0	-ve	*NA*	-ve	*NA*	*NA*	*NA*
	Gyimothy et al. (2005) [31]	-ve	0	-ve	-ve	*NA*	-ve	*NA*	-ve	-ve

NA: Not Applicable

Chapter 5

Refactoring Classification Based on Software Quality Attributes

Based on our literature review, discussed in Chapter 3, we can see that there is a need for guidelines to help the software designer decide which refactorings to apply in order to optimize a software system with regard to certain design goals. Accordingly, we need to study the effect of refactoring by classifying different refactorings based on the quality attributes they aim to improve. This chapter proposes a classification of refactoring methods and refactoring to patterns based on their measurable effect on internal and external software quality attributes.

5.1. Refactoring Methods Classification

In this section, we classify the selected refactoring methods based on internal and external software quality attributes. The classification is done based on simple source code examples which are cited from Fowler's book [29] and refactoring website [10].

5.1.1. Selected Refactoring Methods

We composed a list of refactoring methods that redistribute responsibilities within classes and/or between classes. In addition, the composed list almost covers all the categories of the refactoring catalog. Moreover, it consists of refactoring methods that are among the most popular and most widely used. The following are the selected refactoring methods (for more details see Chapter 2):

- **Consolidate Conditional Expression**: combine sequence of conditional tests with the same result into a single conditional expression and then extract it.
- **Encapsulate Field**: encapsulate public field by making it private and provide accessors to it.
- **Extract Class**: create a new class and move the relevant fields and methods from the old class into the new class.
- **Extract Method**: extract group of statements into a method.
- **Hide Method**: make the method private.
- **Inline Class**: put the class into another class and delete it.
- **Inline Method**: put the method's body into the body of its callers and remove the method.
- **Inline Temp**: replace all references to the temp with the expression.

- **Remove Setting Method**: remove any setting method for a particular field.

- **Replace Assignment with Initialization**: make direct initialization of variable instead of declare the variable first and then assign a value to it.

- **Replace Magic Number with Symbolic Constant**: create a constant and replace the number with it.

- **Reverse Conditional**: reverse the sense of the conditional and reorder the conditional's clauses.

5.1.2. Source Code Examples

This section presents the source code examples for the selected refactoring methods. We used these examples to classify the investigated refactoring methods. The source code examples are cited from Fowler's book [29] and refactoring website [10]. Figure 5-20 to Figure 5-31 provide the source code examples for the investigated refactoring methods.

```
// BEFORE CONSOLIDATE CONDITIONAL EXPRESSION
class Amount{
    public int seniority;
    public int monthsDisabled;
    public boolean isPartTime;

    public double disabilityAmount() {
        if (seniority < 2)
             return 0;
        if (monthsDisabled > 12)
             return 0;
        if (isPartTime)
             return 0;

        // compute the disability amount
    }
    public int getSeniority(){
        return seniority;
    }
    public int getmonthsDisabled(){
        return monthsDisabled;
    }
}

// AFTER CONSOLIDATE CONDITIONAL EXPRESSION
class Amount{
    public int seniority;
    public int monthsDisabled;
    public boolean isPartTime;

    public double disabilityAmount() {

        if (isNotEligibleForDisability())
             return 0;

        // compute the disability amount
    }
    public boolean isNotEligibleForDisability() {
             return ((seniority < 2) || (monthsDisabled > 12)
                     || (isPartTime));
    }
    public int getSeniority(){
        return seniority;
    }
    public int getmonthsDisabled(){
        return monthsDisabled;
    }
}
```

Figure 5-20. Consolidate conditional expression: source code example

```
// BEFORE ENCAPSULATE FIELD
class Customer {

    public String name;
    private String id;

    public String getID() {
        return id;
    }
    public void setID(String arg) {
        id = arg;
    }
}

// AFTER ENCAPSULATE FIELD
class Customer {

    private String name;
    private String id;

    public String getName() {
        return name;
    }
    public void setName(String arg) {
        name = arg;
    }

    public String getID() {
        return id;
    }
    public void setID(String arg) {
        id = arg;
    }
}
```

Figure 5-21. Encapsulate field: source code example

```
// BEFORE EXTRACT CLASS
class Person {
    private String name;
    private String officeAreaCode;
    private String officeNumber;
    public String getName() {
        return name;
    }
    public String getTelephoneNumber() {
        return ("(" + officeAreaCode + ") " + officeNumber);
    }
    public String getOfficeAreaCode() {
        return officeAreaCode;
    }
    public void setOfficeAreaCode(String arg) {
        officeAreaCode = arg;
    }
    public String getOfficeNumber() {
        return officeNumber;
    }
    public void setOfficeNumber(String arg) {
        officeNumber = arg;
    }
}
// AFTER EXTRACT CLASS
class Person {
    private String name;
    private TelephoneNumber officeTelephone = new TelephoneNumber();
    public String getName() {
        return name;
    }
    public String getTelephoneNumber(){
        return officeTelephone.getTelephoneNumber();
    }
    public TelephoneNumber getOfficeTelephone() {
        return officeTelephone;
    }
}
class TelephoneNumber {
    private String number;
    private String areaCode;
    public String getTelephoneNumber() {
        return ("(" + areaCode + ") " + number);
    }
    public String getAreaCode() {
        return areaCode;
    }
    public void setAreaCode(String arg) {
        areaCode = arg;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String arg) {
        number = arg;
    }
}
```

Figure 5-22. Extract class: source code example

```
// BEFORE EXTRACT METHOD
class Extract {

    private String name;

    public void printOwing() {

         Enumeration e = orders.elements();
         double outstanding = 0.0;

         printBanner();

         // calculate outstanding
         while (e.hasMoreElements()) {
              Order each = (Order) e.nextElement();
              outstanding += each.getAmount();
         }

         //print details
         System.out.println ("name:" + name);
         System.out.println ("amount" + outstanding);
   }
}

// AFTER EXTRACT METHOD
class Extract {

    private String name;

    public void printOwing() {

         Enumeration e = orders.elements();
         double outstanding = 0.0;

         printBanner();

         // calculate outstanding
         while (e.hasMoreElements()) {
              Order each = (Order) e.nextElement();
              outstanding += each.getAmount();
         }

         printDetails(outstanding);
    }

    public void printDetails (double outstanding) {
         System.out.println ("name:" + name);
         System.out.println ("amount" + outstanding);
    }
}
```

Figure 5-23. Extract method: source code example

```
// BEFORE HIDE METHOD
class Customer{
    private String customerID;
    private String customerName;
    private int day;

    public Customer(String customerID, String customerName, int day){
        this.customerID = customerID;
        this.customerName = customerName;
        this.day = day;
    }
    public String getCustomerID(){
        return customerID;
    }
    public String getCustomerName(){
        return customerName;
    }
    public int getDay(){
        return day;
    }
    public void setCustomerID(String newCustomerID){
        customerID = newCustomerID ;
    }
    public void setCustomerName(String newCustomerName){
        customerName = newCustomerName;
    }
}

// AFTER HIDE METHOD
class Customer{
   private String customerID;
   private String customerName;
   private int day;

   public Customer(String customerID, String customerName, int day){
        this.customerID = customerID;
        this.customerName = customerName;
        this.day = day;
   }
   private String getCustomerID(){
        return customerID;
   }
   private String getCustomerName(){
        return customerName;
   }
   private int getDay(){
        return day;
   }
   private void setCustomerID(String newCustomerID){
        customerID = newCustomerID ;
   }
   private void setCustomerName(String newCustomerName){
        customerName = newCustomerName;
   }
}
```

Figure 5-24. Hide method: source code example

```
// BEFORE INLINE CLASS
class Person {
    private String name;
    private TelephoneNumber officeTelephone = new TelephoneNumber();
    public String getName() {
        return name;
    }
    public String getTelephoneNumber(){
        return officeTelephone.getTelephoneNumber();
    }
    public TelephoneNumber getOfficeTelephone() {
        return officeTelephone;
    }
}
class TelephoneNumber {
    private String number;
    private String areaCode;
    public String getTelephoneNumber() {
        return ("(" + areaCode + ") " + number);
    }
    public String getAreaCode() {
        return areaCode;
    }
    public void setAreaCode(String arg) {
        areaCode = arg;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String arg) {
        number = arg;
    }
}
// AFTER INLINE CLASS
class Person {
    private String name;
    private String officeAreaCode;
    private String officeNumber;
    public String getName() {
        return name;
    }
    public String getTelephoneNumber() {
        return ("(" + officeAreaCode + ") " + officeNumber);
    }
    public String getOfficeAreaCode() {
        return officeAreaCode;
    }
    public void setOfficeAreaCode(String arg) {
        officeAreaCode = arg;
    }
    public String getOfficeNumber() {
        return officeNumber;
    }
    public void setOfficeNumber(String arg) {
        officeNumber = arg;
    }
}
```

Figure 5-25. Inline class: source code example

```
// BEFORE INLINE METHOD
class Delivery {
   private int numberOfLateDeliveries;

   public int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
   }

   public boolean moreThanFiveLateDeliveries() {
        return numberOfLateDeliveries > 5;
   }

   public int getNumberOfLateDeliveries() {
        return numberOfLateDeliveries;
   }
}

// AFTER INLINE METHOD
class Delivery {
   private int numberOfLateDeliveries;

   public int getRating() {
        return (numberOfLateDeliveries > 5) ? 2 : 1;
   }

   public int getNumberOfLateDeliveries() {
        return numberOfLateDeliveries;
   }
}
```

Figure 5-26. Inline method: source code example

```
// BEFORE INLINE TEMP
class Price{
   private double quantity;
   private double itemPrice;

   public double getPrice() {
         double price = quantity * itemPrice;
         return price;
   }
}

// AFTER INLINE TEMP
class Price{
   private double quantity;
   private double itemPrice;

   public double getPrice() {
         return quantity * itemPrice;
   }
}
```

Figure 5-27. Inline temp: source code example

```
// BEFORE REMOVE SETTING METHOD
class Account {

   private String id;
   private String name;

   public Account (String id, String name) {
        setId(id);
        setName(name);
   }
   public void setId (String arg) {
        id = arg;
   }
   public String getId() {
        return id;
   }
   public void setName (String arg) {
        name = arg;
   }
   public String getName() {
        return name;
   }
}

// AFTER REMOVE SETTING METHOD
class Account {

   private String id;
   private String name;

   public Account (String arg1, String arg2) {
        id = arg1;
        name = arg2;
   }
   public String getId() {
        return id;
   }
   public String getName() {
        return name;
   }
}
```

Figure 5-28. Remove setting method: source code example

```
// BEFORE REPLACE ASSIGNMENT WITH INITIALIZATION
class Price{
   private int quantity;
   private int itemPrice;

   public double getPrice() {
        int basePrice;
        basePrice = quantity * itemPrice;
        double discountFactor;
        if (basePrice > 1000)
             discountFactor = 0.95;
        else
             discountFactor = 0.98;
        return basePrice * discountFactor;
   }
}

// AFTER REPLACE ASSIGNMENT WITH INITIALIZATION
class Price{
   private int quantity;
   private int itemPrice;

   public double getPrice() {
        int basePrice = quantity * itemPrice;
        double discountFactor;
        if (basePrice > 1000)
             discountFactor = 0.95;
        else
             discountFactor = 0.98;
        return basePrice * discountFactor;
   }
}
```

Figure 5-29. Replace assignment with initialization: source code example

```
// BEFORE REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT
class Energy {

   public double potentialEnergy(double mass, double height) {
        return mass * 9.81 * height;
   }
}

// AFTER REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT
class Energy {

   static final double GRAVITATIONAL_CONSTANT = 9.81;

   public double potentialEnergy(double mass, double height) {
        return mass * GRAVITATIONAL_CONSTANT * height;
    }
}
```

Figure 5-30. Replace magic number with symbolic constant: source code example

```
// BEFORE REVERSE CONDITIONAL
class Charge {
   double charge;
   double quantity;
   double winterRate;
   double winterServiceCharge;
   double summerRate;
   Date date;
   final String SUMMER_START = "1/6/2008";
   final String SUMMER_END = "31/8/2008";
   public void calculateRate(){
       if ( !isSummer(date))
           charge = winterCharge(quantity);
       else
           charge = summerCharge (quantity);
   }

   private boolean notSummer(Date date) {
        return date.before (SUMMER_START) || date.after(SUMMER_END);
   }
   private double summerCharge(double quantity) {
        return quantity * summerRate;
   }
   private double winterCharge(double quantity) {
        return quantity * winterRate + winterServiceCharge;
   }
}

// AFTER REVERSE CONDITIONAL
class Charge {
   double charge;
   double quantity;
   double winterRate;
   double winterServiceCharge;
   double summerRate;
   Date date;
   final String SUMMER_START = "1/6/2008";
   final String SUMMER_END = "31/8/2008";
   public void calculateRate(){
       if (isSummer(date))
           charge = summerCharge (quantity);
       else
           charge = winterCharge(quantity);
   }
   private boolean notSummer(Date date) {
        return date.before (SUMMER_START) || date.after(SUMMER_END);
   }
   private double summerCharge(double quantity) {
        return quantity * summerRate;
   }
   private double winterCharge(double quantity) {
        return quantity * winterRate + winterServiceCharge;
   }
}
```

Figure 5-31. Reverse conditional: source code example

5.1.3. Classification of Refactoring Methods Based on Internal Quality Attributes

In this section, we propose a classification of refactoring methods based on their measurable effect on the internal quality metrics described in Chapter 4. The *Understand for Java* metrics tool [13] and the *Metamata* metrics tool [9] were used to collect the internal quality metrics for the source code examples before and after applying the investigated refactoring methods. Table 5-3 provides the measurement results for the source code examples before and after applying the investigated refactoring methods. In order to explain why the internal quality metric increases or decreases as a result of applying a refactoring method, we need to analyze the effect of each refactoring method on the internal quality metrics as follows:

- **Consolidate Conditional Expression**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it reduces the value of LOC since it combines a series of conditional checks into a single conditional check. Moreover, it increases the value of RFC, WMC, NOM, and LCOM metrics as it extracts the single conditional check into a new method. In summary, "Consolidate Conditional Expression" increases the size of the class in terms of number of methods (RFC, WMC, and NOM), reduces the number of source code statements (LOC), and makes the class less cohesive as it assigns more responsibilities to it.

Table 5-3. Measurement results for source code examples before and after applying refactoring methods

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Consolidate Conditional Expression	Before	Amount.java	1	0	0	3	0	3	3	19	0
	After	Amount.java	1	0	0	4	0	4	4	18	2
Encapsulate Field	Before	Customer.java	1	0	1	2	1	2	2	10	0
	After	Customer.java	1	0	1	4	1	4	4	16	2
Extract Class	Before	Person.java	1	0	1	6	1	6	6	23	3
		TelephoneNumber.java	-	-	-	-	-	-	-	-	-
	After	Person.java	1	0	2	4	2	3	3	13	1
		TelephoneNumber.java	1	0	1	5	1	5	5	20	0
Extract Method	Before	Extract.java	1	0	3	2	1	2	1	15	0
	After	Extract.java	1	0	3	3	1	3	2	18	1
Hide Method	Before	Customer.java	1	0	1	5	1	5	5	26	6
	After	Customer.java	1	0	1	5	1	5	5	26	6

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Inline Class	Before	Person.java	1	0	2	4	2	3	3	13	1
		TelephoneNumber.java	1	0	1	5	1	5	5	20	0
	After	Person.java	1	0	1	6	1	6	6	23	3
		TelephoneNumber.java	-	-	-	-	-	-	-	-	-
Inline Method	Before	Delivery.java	1	0	0	3	0	3	3	12	1
	After	Delivery.java	1	0	0	2	0	2	2	9	0
Inline Temp	Before	Price.java	1	0	0	1	0	1	1	8	0
	After	Price.java	1	0	0	1	0	1	1	7	0
Remove Setting Method	Before	Account.java	1	0	1	4	1	4	4	20	2
	After	Account.java	1	0	1	2	1	2	2	14	1
Replace Assignment with Initialization	Before	Price.java	1	0	0	1	0	2	1	14	0
	After	Price.java	1	0	0	1	0	2	1	13	0
Replace Magic Number with Symbolic Constant	Before	Energy.java	1	0	0	1	0	1	1	5	0
	After	Energy.java	1	0	0	1	0	1	1	6	0
Reverse Conditional	Before	Charge.java	1	0	1	4	1	5	4	26	6
	After	Charge.java	1	0	1	4	1	5	4	26	6

- **Encapsulate Field**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics as it provides accessors (setter and getter) methods to a field (attribute). In summary, "Encapsulate Field" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.

- **Extract Class**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. However, it increases the value of CBO and FOUT metrics since the old class is coupled to the extracted class. Additionally, it reduces the value of RFC, WMC, NOM, LOC, and LCOM metrics of the old class as it moves the relevant fields and methods from the old class into the extracted class. However, it increases the number of classes in the system as it extracts new class. In summary, "Extract Class" reduces the size of the old class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), increases the coupling between the classes (CBO, FOUT), makes the old class more cohesive as it reduces the responsibilities assigned to it, and increases the number of classes in the system.

- **Extract Method**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes

of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics as it extracts group of statements into a new method. In summary, "Extract Method" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.

- **Hide Method**: changes the visibility of a method from public to private if the method is not used by other classes. Therefore, it does not affect any of the investigated internal quality metrics.

- **Inline Class**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. However, it reduces the value of CBO and FOUT metrics since class 'A' is placed into another class 'B' and the coupling between them will be removed as class 'A' is deleted. Additionally, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics as it moves the relevant fields and methods from class 'A' to class 'B'. However, it reduces the number of classes in the system as it deletes class 'A'. In summary, "Inline Class" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), reduces the coupling between the classes (CBO, FOUT), makes the class less cohesive as it assigns more responsibilities to it, and reduces the number of classes in the system.

- **Inline Method**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it reduces the value of RFC, WMC, NOM, LOC, and LCOM metrics since method ‘A’ is placed into method ‘B’ and method ‘A’ is removed. In summary, “Inline Method” reduces the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class more cohesive as it reduces the responsibilities assigned to it.

- **Inline Temp**: replaces all references to the temporary variable with an expression and removes the temporary variable. Therefore, it does not affect any of the investigated internal quality metrics except it reduces the source code statements (LOC).

- **Remove Setting Method**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it reduces the value of RFC, WMC, NOM, LOC, and LCOM metrics since it removes any setting method for a particular filed (attribute). In summary, “Remove Setting Method” reduces the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class more cohesive as it reduces the responsibilities assigned to it.

- **Replace Assignment with Initialization**: makes direct initialization of variable instead of declare the variable and then assign a value to it. Therefore, it does not affect any of the investigated internal quality metrics except it reduces the source code statements (LOC).

- **Replace Magic Number with Symbolic Constant**: creates a constant and replaces the number with it. Therefore, it does not affect any of the investigated internal quality metrics except it increases the source code statements (LOC).

- **Reverse Conditional**: reverses the sense of the conditional and reorder the conditional's clauses. Therefore, it does not affect any of the investigated internal quality metrics.

Based on the above analysis and the measurement results presented in Table 5-3, we can classify the investigated refactoring methods according to the internal quality metrics they affect. For example, “Encapsulate Field” and “Extract Method” have the same effect on the internal quality metrics i.e. they increase the metrics value of RFC, WMC, NOM, LOC, and LCOM while they not change the metrics value of DIT, NOC, CBO, and FOUT. Table 5-4 presents the classification of the investigated refactoring methods based on the internal software quality metrics, where “↑” symbol represents an increase in a metric value, “↓” symbol represents a decrease in a metric value, and “-” symbol represents no change in a metric value.

Table 5-4. Refactoring methods classification based on internal software quality attributes

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field	-	-	-	↑	-	↑	↑	↑	↑
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Reverse Conditional	-	-	-	-	-	-	-	-	-
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Consolidate Conditional Expression	-	-	-	↑	-	↑	↑	↓	↑
Extract Class	-	-	↑	↓	↑	↓	↓	↓	↓
Inline Class	-	-	↓	↑	↓	↑	↑	↑	↑
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

5.1.4. Classification of Refactoring Methods Based on External Quality Attributes

In this section, we propose a classification of refactoring methods based on the external quality attributes described in Chapter 4. As it is explained earlier in Chapter 4, the external quality attributes (adaptability, completeness, maintainability, understandability, reusability, testability, and reliability) can be assessed using internal quality metrics.

In order to classify the investigated refactoring methods according to the external quality attributes, we rely on the findings of the existing research studies that show a correlation between external quality attributes and internal quality metrics. More specifically, we rely on the results found by (*i*) Dandashi [26] to assess adaptability, completeness, maintainability, understandability, and reusability (*ii*) Bruntink and Deursen [24] to estimate the testing effort and (*iii*) Basili et al. [14], Tang et al. [50], and Gyimo´thy et al. [31] to predict the reliability of software systems. Therefore, the classification is done by mapping the changes in the internal quality metrics to the external quality attributes based on these research studies which are discussed in details in Chapter 4.

For example, to see whether "Encapsulate Field" increases (improves) the adaptability or not, we can see from Table 5-4 that "Encapsulate Field" increases the metrics value of RFC, WMC, NOM, LOC, and LCOM. The changes in these metrics values are mapped to the adaptability based on Dandashi [26] study that shows a negative correlation between the adaptability and RFC metric and a positive correlation between

the adaptability and WMC and LOC metrics. Therefore, “Encapsulate Field” increases (improves) the adaptability as WMC and LOC metrics increase.

Table 5-5 presents the classification of the investigated refactoring methods based on the external software quality attributes, where “↑” symbol represents an increase (improve) in external quality attribute except for the testability (testing effort) it means an impair, “↓” symbol represents a decrease (impair) in external quality attribute except for the testability (testing effort) it means an improve, and “-” symbol represents no change in external quality attribute.

Table 5-5. Refactoring methods classification based on external software quality attributes

Refactoring Method	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability	Reliability
Inline Method	↓	↓	↓	↓	↓	↓	↑
Remove Setting Method	↓	↓	↓	↓	↓	↓	↑
Inline Temp	↓	↓	↓	↓	↓	↓	↑
Replace Assignment with Initialization	↓	↓	↓	↓	↓	↓	↑
Extract Class	↓	↓	↓	↓	↓	↓	↑
Encapsulate Field	↑	↑	↑	↑	↑	↑	↓
Extract Method	↑	↑	↑	↑	↑	↑	↓
Inline Class	↑	↑	↑	↑	↑	↑	↓
Replace Magic Number with Symbolic Constant	↑	↑	↑	↑	↑	↑	↓
Hide Method	-	-	-	-	-	-	-
Reverse Conditional	-	-	-	-	-	-	-
Consolidate Conditional Expression	↓	↓	↓	↓	↓	↑	↓

5.2. Refactoring to Patterns Classification

In this section, we classify the selected refactoring to patterns based on internal and external software quality attributes. The classification is done based on real-world software projects which are cited from Kerievsky's book [39].

5.2.1. Selected Refactoring to Patterns

We composed a list of refactoring to patterns that almost covers all the categories of refactoring to patterns catalog. Moreover, the composed list consists of refactoring to patterns that are among the most popular and most widely used. The following are the selected refactoring to patterns (for more details see Chapter 2):

- **Chain Constructors**: chain the constructors together to get the least amount of duplicate code.

- **Compose Method**: transform a method's logic into a small number of steps at the same level of detail.

- **Form Template Method**: generalize the methods in subclasses by extracting their steps into methods with identical signatures, then pull up the generalized methods to form a Template Method.

- **Introduce Null Object**: replace the null logic with a null object which provides the appropriate null behavior.

- **Replace Conditional Dispatcher with Command**: replace the conditional logic with code to fetch and execute commands.

- **Replace Constructors with Creation Methods**: replace the constructors with creation methods that return object instances.

- **Unify Interfaces**: provide a superclass/interface with same interface as a subclass.

5.2.2. Source Code Examples

This section presents the source code examples for the selected refactoring to patterns. We used these examples to classify the investigated refactoring to patterns. The source code examples are part from real-world software projects which are cited from Kerievsky's book [39]. Figure 5-32 to Figure 5-38 provide the source code examples for the investigated refactoring to patterns.

```
// BEFORE CHAIN CONSTRUCTORS
public class Loan {

    public float notional;
    public int rating;
    public Date maturity;
    public Date expiry;
    public float outstanding;
    public CapitalStrategy strategy;

    public Loan(float notional, float outstanding, int rating, Date expiry) {
       this.strategy = new TermROC();
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
    }

    public Loan(float notional, float outstanding, int rating, Date expiry,
                Date maturity) {
       this.strategy = new RevolvingTermROC();
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }


    public Loan(CapitalStrategy strategy, float notional, float outstanding,
                int rating, Date expiry, Date maturity) {
       this.strategy = strategy;
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }
}

// AFTER CHAIN CONSTRUCTORS
public class Loan {

    public float notional;
    public int rating;
    public Date maturity;
    public Date expiry;
    public float outstanding;
    public CapitalStrategy strategy;

    public Loan(float notional, float outstanding, int rating, Date expiry) {
        this(new TermROC(), notional, outstanding, rating, expiry, null);
    }
```

```
 public Loan(float notional, float outstanding, int rating, Date expiry,
                Date maturity) {
        this(new RevolvingTermROC(), notional, outstanding, rating, expiry,
             maturity);
    }

public Loan(CapitalStrategy strategy, float notional, float outstanding,
                int rating, Date expiry, Date maturity) {
       this.strategy = strategy;
       this.notional = notional;
       this.outstanding = outstanding;
       this.rating = rating;
       this.expiry = expiry;
       this.maturity = maturity;
    }
}
```

Figure 5-32. Chain constructors: source code example

```
// BEFORE COMPOSE METHOD
public class List {

    private boolean readOnly;
    private int size;
    private Object[] elements;

    public void add(Object element) {
          if (!readOnly) {
             int newSize = size + 1;
             if (newSize > elements.length) {
                Object[] newElements =
                   new Object[elements.length + 10];
                for (int i = 0; i < size; i++)
                   newElements[i] = elements[i];
                elements = newElements;
             }
             elements[size++] = element;
          }
    }
}

// AFTER COMPOSE METHOD
public class List {

    private boolean readOnly;
    private int size;
    private Object[] elements;

    public void add(Object element) {
          if (readOnly)
             return;
          if (atCapacity())
             grow();
          addElement(element);
    }
    private boolean atCapacity() {
        return (size + 1) > elements.length;
    }
    private void grow() {
        Object[] newElements = new Object[elements.length + 10];
        for (int i = 0; i < size; i++)
            newElements[i] = elements[i];
        elements = newElements;
    }
    private void addElement(Object element) {
        elements[size++] = element;
    }
}
```

Figure 5-33. Compose method: source code example

```
// BEFORE FORM TEMPLATE METHOD
public abstract class CapitalStrategy {

    public abstract double capital(Loan loan);
}

public class CapitalStrategyTermLoan extends CapitalStrategy{

    public double capital(Loan loan) {
          return loan.getCommitment() * duration(loan) * riskFactorFor(loan);
    }
}

public class CapitalStrategyAdvisedLine extends CapitalStrategy{

    public double capital(Loan loan) {
          return loan.getCommitment() * loan.getUnusedPercentage() *
                 duration(loan) * riskFactorFor(loan);
    }
}

// AFTER FORM TEMPLATE METHOD
public abstract class CapitalStrategy {

    public double capital(Loan loan) {

        return riskAmountFor(loan) * duration(loan) * riskFactorFor(loan);
    }

    public abstract double riskAmountFor(Loan loan);
}

public class CapitalStrategyTermLoan extends CapitalStrategy{

    private double riskAmountFor(Loan loan) {

        return loan.getCommitment();
    }
}

public class CapitalStrategyAdvisedLine extends CapitalStrategy{

    private double riskAmountFor(Loan loan) {

        return loan.getCommitment() * loan.getUnusedPercentage();
    }
}
```

Figure 5-34. Form template method: source code example

```
// BEFORE INTRODUCE NULL OBJECT
public class MouseEventHandler {

    public MouseEventHandler(){}
    public boolean mouseMove(Event event, int x, int y) {}
    public boolean mouseDown(Event event, int x, int y) {}
    public boolean mouseUp(Event event, int x, int y) {}
    public boolean mouseExit(Event event, int x, int y) {}
}

public class NavigationApplet extends Applet {

    public boolean mouseMove(Event event, int x, int y) {
        if (mouseEventHandler != null)
           return mouseEventHandler.mouseMove(graphicsContext, event, x, y );
        return true;
    }
    public boolean mouseDown(Event event, int x, int y) {
        if (mouseEventHandler != null)
           return mouseEventHandler.mouseDown(graphicsContext, event, x, y );
        return true;
    }
    public boolean mouseUp(Event event, int x, int y) {
        if (mouseEventHandler != null)
           return mouseEventHandler.mouseUp(graphicsContext, event, x, y );
        return true;
    }
    public boolean mouseExit(Event event, int x, int y) {
        if (mouseEventHandler != null)
           return mouseEventHandler.mouseExit(graphicsContext, event, x, y );
        return true;
    }
}

// AFTER INTRODUCE NULL OBJECT
public class MouseEventHandler {

    public MouseEventHandler(){}
    public boolean mouseMove(Event event, int x, int y) {}
    public boolean mouseDown(Event event, int x, int y) {}
    public boolean mouseUp(Event event, int x, int y) {}
    public boolean mouseExit(Event event, int x, int y) {}
}

public class NullMouseEventHandler extends MouseEventHandler {

    public NullMouseEventHandler(Context context) {
        super(null);
    }
    public boolean mouseMove(MetaGraphicsContext mg, Event e, int x, int y) {
        return true;
    }
    public boolean mouseDown(MetaGraphicsContext mg, Event e, int x, int y) {
        return true;
    }
    public boolean mouseUp(MetaGraphicsContext mg, Event e, int x, int y) {
        return true;
    }
```

```
    public boolean mouseExit(MetaGraphicsContext mg, Event e, int x, int y) {
        return true;
    }
}

public class NavigationApplet extends Applet {

   private MouseEventHandler mouseEventHandler = new NullMouseEventHandler();

   public boolean mouseMove(Event event, int x, int y) {
        return mouseEventHandler.mouseMove(graphicsContext, event, x, y );
   }
   public boolean mouseDown(Event event, int x, int y) {
        return mouseEventHandler.mouseDown(graphicsContext, event, x, y );
   }
   public boolean mouseUp(Event event, int x, int y) {
        return mouseEventHandler.mouseUp(graphicsContext, event, x, y );
   }
   public boolean mouseExit(Event event, int x, int y) {
        return mouseEventHandler.mouseExit(graphicsContext, event, x, y );
   }
}
```

Figure 5-35. Introduce null object: source code example

```
// BEFORE REPLACE CONDITIONAL DISPATCHER WITH COMMAND
public class CatalogApp {

    private HandlerResponse executeActionAndGetResponse(String actionName, Map
                                                              parameters){
        if (actionName.equals(NEW_WORKSHOP)) {
          String nextWorkshopID = workshopManager.getNextWorkshopID();
          StringBuffer newWorkshopContents =
            workshopManager.createNewFileFromTemplate(nextWorkshopID,
              workshopManager.getWorkshopDir(),
              workshopManager.getWorkshopTemplate());
          workshopManager.addWorkshop(newWorkshopContents);
          parameters.put("id",nextWorkshopID);
          executeActionAndGetResponse(ALL_WORKSHOPS, parameters);
        } else if (actionName.equals(ALL_WORKSHOPS)) {
          XMLBuilder allWorkshopsXml = new XMLBuilder("workshops");
          WorkshopRepository repository =
            workshopManager.getWorkshopRepository();
          Iterator ids = repository.keyIterator();
          while (ids.hasNext()) {
            String id = (String)ids.next();
            Workshop workshop = repository.getWorkshop(id);
            allWorkshopsXml.addBelowParent("workshop");
            allWorkshopsXml.addAttribute("id", workshop.getID());
            allWorkshopsXml.addAttribute("name", workshop.getName());
            allWorkshopsXml.addAttribute("status", workshop.getStatus());
            allWorkshopsXml.addAttribute("duration",
                                              workshop.getDurationAsString());
          }
          String formattedXml = getFormattedData(allWorkshopsXml.toString());
          return new HandlerResponse(new StringBuffer(formattedXml),
                                                    ALL_WORKSHOPS_STYLESHEET);
        }
        //many more "else if" statements
    }
}

// AFTER REPLACE CONDITIONAL DISPATCHER WITH COMMAND
public class AllWorkshopsHandler extends Handler{

    private CatalogApp catalogApp;
    private static String ALL_WORKSHOPS_STYLESHEET="allWorkshops.xsl";
    private PrettyPrinter prettyPrinter = new PrettyPrinter();

    public AllWorkshopsHandler(CatalogApp catalogApp) {
        super(catalogApp);
    }
    public HandlerResponse execute(Map parameters) throws Exception {
        return new HandlerResponse(new
      StringBuffer(prettyPrint(allWorkshopsData())), ALL_WORKSHOPS_STYLESHEET);
    }
    private String allWorkshopsData(){ }

    private String prettyPrint(String buffer) {
        return prettyPrinter.format(buffer);
    }
}
```

```
public class NewWorkshopHandler extends Handler{

    private CatalogApp catalogApp;

    public NewWorkshopHandler(CatalogApp catalogApp) {
        super(catalogApp);
    }
    public HandlerResponse execute(Map parameters) throws Exception {

        String nextWorkshopID = workshopManager().getNextWorkshopID();
        StringBuffer newWorkshopContents =
                        WorkshopManager().createNewFileFromTemplate(
                        nextWorkshopID, workshopManager().getWorkshopDir(),
                        workshopManager().getWorkshopTemplate());
        workshopManager().addWorkshop(newWorkshopContents);
        parameters.put("id", nextWorkshopID);
        catalogApp.executeActionAndGetResponse(ALL_WORKSHOPS, parameters);
    }
    private WorkshopManager workshopManager() {
        return catalogApp.getWorkshopManager();
    }
}

public abstract class Handler {

    protected CatalogApp catalogApp;

    public Handler(CatalogApp catalogApp) {
        this.catalogApp = catalogApp;
    }
    public abstract HandlerResponse execute(Map parameters) throws Exception;
}

public class CatalogApp {

    private Map handlers;

    public CatalogApp() {
        createHandlers();
    }
    public void createHandlers() {
        handlers = new HashMap();
        handlers.put(NEW_WORKSHOP, new NewWorkshopHandler(this));
        handlers.put(ALL_WORKSHOPS, new AllWorkshopsHandler(this));
    }
    public HandlerResponse executeActionAndGetResponse(String handlerName, Map
                                                    parameters) throws Exception{
        Handler handler = lookupHandlerBy(handlerName);
        return handler.execute(parameters);
    }
    private Handler lookupHandlerBy(String handlerName) {
        return (Handler)handlers.get(handlerName);
    }
}
```

Figure 5-36. Replace conditional dispatcher with command: source code example

```
// BEFORE REPLACE CONSTRUCTORS WITH CREATION METHODS
public class Loan {

  public double commitment;
  public int riskRating;
  public Date maturity;
  public Date expiry;
  public double outstanding;
  public int customerRating;

  public Loan(double commitment, int riskRating, Date maturity, Date expiry){
      this(commitment, 0.00, riskRating, maturity, expiry);
  }
  public Loan(double commitment, double outstanding, int customerRating, Date
              maturity, Date expiry) {
      this(null, commitment, outstanding, customerRating, maturity, expiry);
  }
  public Loan(CapitalStrategy capitalStrategy, double commitment, double
              outstanding, int riskRating, Date maturity, Date expiry) {
      this.commitment = commitment;
      this.outstanding = outstanding;
      this.riskRating = riskRating;
      this.maturity = maturity;
      this.expiry = expiry;
      this.capitalStrategy = capitalStrategy;
      if (capitalStrategy == null) {
         if (expiry == null)
            this.capitalStrategy = new CapitalStrategyTermLoan();
         else if (maturity == null)
            this.capitalStrategy = new CapitalStrategyRevolver();
         else
            this.capitalStrategy = new CapitalStrategyRCTL();
      }
  }
}

public class CapitalCalculationTests {

   public double commitment;
   public int riskRating;
   public Date maturity;
   public double outstanding;
   CapitalStrategy riskAdjustedCapitalStrategy;

   public void testTermLoanNoPayments() {
       Loan termLoan = new Loan (commitment, riskRating, maturity);
   }
   public void testTermLoanWithRiskAdjustedCapitalStrategy() {
       Loan termLoan = new Loan (riskAdjustedCapitalStrategy, commitment,
                                 outstanding, riskRating, maturity);
   }
}
```

```
// AFTER REPLACE CONSTRUCTORS WITH CREATION METHODS
public class Loan {
   public double commitment;
   public int riskRating;
   public Date maturity;
   public Date expiry;
   public double outstanding;
   public int customerRating;
   private Loan(double commitment, double outstanding, int customerRating,
                Date maturity, Date expiry) {
       this(null, commitment, outstanding, customerRating, maturity, expiry);
   }
   private Loan(CapitalStrategy capitalStrategy, double commitment, double
                outstanding, int riskRating, Date maturity, Date expiry) {
       this.commitment = commitment;
       this.outstanding = outstanding;
       this.riskRating = riskRating;
       this.maturity = maturity;
       this.expiry = expiry;
       this.capitalStrategy = capitalStrategy;
       if (capitalStrategy == null) {
          if (expiry == null)
             this.capitalStrategy = new CapitalStrategyTermLoan();
          else if (maturity == null)
             this.capitalStrategy = new CapitalStrategyRevolver();
          else
             this.capitalStrategy = new CapitalStrategyRCTL();
       }
   }
   public static Loan createTermLoan(double commitment, int riskRating, Date
                                     maturity) {
         return new Loan(commitment, 0.00, riskRating, maturity, null);
   }
   public static Loan createTermLoan(CapitalStrategy
      riskAdjustedCapitalStrategy, double commitment, double outstanding, int
      riskRating, Date maturity) {

         return new Loan(riskAdjustedCapitalStrategy, commitment, outstanding,
                         riskRating, maturity, null);
   }
}
public class CapitalCalculationTests {
   public double commitment;
   public int riskRating;
   public Date maturity;
   public double outstanding;
   CapitalStrategy riskAdjustedCapitalStrategy;
   public void testTermLoanNoPayments() {
       Loan termLoan = Loan.createTermLoan(commitment, riskRating, maturity);
   }
   public void testTermLoanWithRiskAdjustedCapitalStrategy() {
       Loan termLoan = Loan.createTermLoan(riskAdjustedCapitalStrategy,
                                commitment, outstanding, riskRating, maturity);
   }
}
```

Figure 5-37. Replace constructors with creation methods: source code example

```
// BEFORE UNIFY INTERFACES
public class AbstractNode {

    public void setElement(){
         // implementation details…
    }
    public void setParent(){
         // implementation details…
    }
}

public class StringNode extends AbstractNode {

    public void setElement(){
           // implementation details…
    }
    public void setParent(){
         // implementation details…
    }
    public void accept() {
        // implementation details…
    }
}

// AFTER UNIFY INTERFACES
public class AbstractNode {

    public void setElement(){
         // implementation details…
    }
    public void setParent(){
         // implementation details…
    }
    public void accept() {}
}

public class StringNode extends AbstractNode {

    public void setElement(){
           // implementation details…
    }
    public void setParent(){
         // implementation details…
    }
    public void accept() {
        // implementation details…
    }
}
```

Figure 5-38. Unify interfaces: source code example

5.2.3. Classification of Refactoring to Patterns Based on Internal Quality Attributes

In this section, we propose a classification of refactoring to patterns based on their measurable effect on the internal quality metrics described in Chapter 4. The *Understand for Java* metrics tool [13] and the *Metamata* metrics tool [9] were used to collect the internal quality metrics for the source code examples before and after applying the investigated refactoring to patterns. Table 5-6 provides the measurement results for the source code examples before and after applying the investigated refactoring to patterns. In order to explain why the internal quality metric increases or decreases as a result of applying refactoring to patterns method, we need to analyze the effect of each refactoring to patterns method on the internal quality metrics as follows:

- **Chain Constructors**: chains the constructors together to reduce the amount of duplicate code. Therefore, it does not affect any of the investigated internal quality metrics except it reduces the source code statements (LOC).

Table 5-6. Measurement results for source code examples before and after applying refactoring to patterns

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Chain Constructors	Before	Loan.java	1	0	0	0	0	0	0	32	0
	After	Loan.java	1	0	0	0	0	0	0	23	0
Compose Method	Before	List.java	1	0	0	1	0	1	1	19	0
	After	List.java	1	0	0	4	0	4	4	26	3
Form Template Method	Before	CapitalStrategy.java	1	2	0	1	0	0	1	4	0
		CapitalStrategyAdvisedLine.java	2	0	0	1	0	1	1	6	0
		CapitalStrategyTermLoan.java	2	0	0	1	0	1	1	6	0
	After	CapitalStrategy.java	1	2	0	2	0	1	2	7	1
		CapitalStrategyAdvisedLine.java	2	0	0	1	0	1	1	6	0
		CapitalStrategyTermLoan.java	2	0	0	1	0	1	1	6	0
Introduce Null Object	Before	MouseEventHandler.java	1	0	0	4	0	4	4	8	6
		NavigationApplet.java	2	0	0	4	0	4	4	23	0
	After	MouseEventHandler.java	1	1	0	4	0	4	4	8	6
		NavigationApplet.java	2	0	0	4	0	4	4	16	0
		NullMouseEventHandler.java	2	0	0	4	0	4	4	18	6

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Conditional Dispatcher with Command	Before	AllWorkshopsHandler.java	-	-	-	-	-	-	-	-	-
		CatalogApp.java	1	0	0	1	1	1	1	31	0
		Handler.java	-	-	-	-	-	-	-	-	-
		NewWorkshopHandler.java	-	-	-	-	-	-	-	-	-
	After	AllWorkshopsHandler.java	2	0	3	3	3	3	3	16	3
		CatalogApp.java	1	0	3	3	3	3	3	19	1
		Handler.java	1	2	2	1	2	0	1	8	0
		NewWorkshopHandler.java	2	0	4	2	4	2	2	20	0
Replace Constructors with Creation Methods	Before	CapitalCalculationTests.java	1	0	2	2	2	2	2	14	0
		Loan.java	1	0	1	0	1	0	0	34	0
	After	CapitalCalculationTests.java	1	0	2	2	2	2	2	14	0
		Loan.java	1	0	1	2	1	2	2	37	1
Unify Interfaces	Before	AbstractNode.java	1	1	0	2	0	2	2	7	1
		StringNode.java	2	0	0	3	0	3	3	9	3
	After	AbstractNode.java	1	1	0	3	0	3	3	8	3
		StringNode.java	2	0	0	3	0	3	3	9	3

- **Compose Method**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics since it composes methods (extracts several groups of statements into new methods). In summary, "Compose Method" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.

- **Form Template Method**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics since the parent class generalizes the methods in the subclasses by extracting their steps into methods with identical signatures and then pulls up the generalized methods to form a Template Method. In summary, "Form Template Method" increases the size of the parent class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the parent class less cohesive as it assigns more responsibilities to it.

- **Introduce Null Object**: replaces the null logic with a null object which provides the appropriate null behavior. The null object is created by extracting subclass on

the source class. Therefore, it increases NOC metric as the source class has new subclass and consequently it increases the number of classes in the system. Additionally, it reduces LOC metric of the target class as it removes the alternative actions if the object is null. In summary, “Introduce Null Object” increases the number of subclasses (NOC) of the source class, reduces the size of the target class in terms of source code statements (LOC), and increases the number of classes in the system.

- **Replace Conditional Dispatcher with Command**: creates a command for each action by creating new class (concrete command) for each command to handle the action. Then, it creates an interface or abstract class that declares an execution method. After that, on the class that contains the conditional dispatcher, it defines and populates a command map that contains instances of each concrete command. Therefore, “Replace Conditional Dispatcher with Command” does not affect DIT and NOC metrics because it neither inherits a class nor inherited by other classes. However, it increases the value of CBO and FOUT metrics since the class that contains the conditional dispatcher is coupled to the abstract class that declares an execution method. Additionally, it increases the value of RFC, WMC, NOM, and LCOM metrics of the class that contains the conditional dispatcher as it defines and populates a command map. Moreover, it reduces LOC metric of the class that contains the conditional dispatcher as it moves the actions implementation to the concrete command. Furthermore, the number of classes in the system is increased

since a new class is created for each command to handle the action. In summary, “Replace Conditional Dispatcher with Command” increases the size of the class in terms of number of methods (RFC, WMC, and NOM), increases the coupling between the classes (CBO, FOUT), reduces the source code statement (LOC), makes the class less cohesive as it assigns more responsibilities to it, and increases the number of classes in the system.

- **Replace Constructors with Creation Methods**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics since it replaces the constructors with creation methods that return object instances. In summary, “Replace Constructors with Creation Methods” increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.

- **Unify Interfaces**: does not affect DIT and NOC metrics because it neither inherits a class nor creates subclasses. Furthermore, it does not use methods or attributes of other classes and consequently it does not have effect on CBO and FOUT metrics. However, it increases the value of RFC, WMC, NOM, LOC, and LCOM metrics of the superclass since it adds to the superclass copies of all public methods on the subclass that are missing on the superclass. In summary, “Unify

Interfaces" increases the size of the superclass class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the superclass class less cohesive as it assigns more responsibilities to it.

Based on the above analysis and the measurement results presented in Table 5-6, we can classify the investigated refactoring to patterns according to the internal quality metrics they affect. For example, "Compose Method", "Form Template Method", "Replace Constructors with Creation Methods", and "Unify Interfaces" have the same effect on the internal quality metrics i.e. they increase the metrics value of RFC, WMC, NOM, LOC, and LCOM while they not change the metrics value of DIT, NOC, CBO, and FOUT. Table 5-7 presents the classification of the investigated refactoring to patterns based on the internal software quality metrics, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 5-7. Refactoring to patterns classification based on internal software quality attributes

Refactoring to Patterns	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Compose Method	-	-	-	↑	-	↑	↑	↑	↑
Form Template Method	-	-	-	↑	-	↑	↑	↑	↑
Replace Constructors with Creation Methods	-	-	-	↑	-	↑	↑	↑	↑
Unify Interfaces	-	-	-	↑	-	↑	↑	↑	↑
Chain Constructors	-	-	-	-	-	-	-	↓	-
Introduce Null Object	-	↑	-	-	-	-	-	↓	-
Replace Conditional Dispatcher with Command	-	-	↑	↑	↑	↑	↑	↓	↑

5.2.4. Classification of Refactoring to Patterns Based on External Quality Attributes

In this section, we propose a classification of refactoring to patterns based on the external quality attributes described in Chapter 4. To achieve this, we used the same approach described in Section 5.1.4 for refactoring methods classification. Table 5-8 presents the classification of the investigated refactoring to patterns based on the external software quality attributes, where "↑" symbol represents an increase (improve) in external quality attribute except for the testability (testing effort) it means an impair, "↓" symbol represents a decrease (impair) in external quality attribute except for the testability (testing effort) it means an improve, and "-" symbol represents no change in external quality attribute.

Table 5-8. Refactoring to patterns classification based on external software quality attributes

Refactoring to Patterns	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability	Reliability
Compose Method	↑	↑	↑	↑	↑	↑	↓
Form Template Method	↑	↑	↑	↑	↑	↑	↓
Replace Constructors with Creation Methods	↑	↑	↑	↑	↑	↑	↓
Unify Interfaces	↑	↑	↑	↑	↑	↑	↓
Chain Constructors	↓	↓	↓	↓	↓	↓	↑
Introduce Null Object	↓	↓	↓	↓	↓	↓	↑
Replace Conditional Dispatcher with Command	↓	↓	↓	↓	↓	↑	↓

Chapter 6

Empirical Validations

In the previous chapter, we classified refactoring methods and refactoring to patterns based on internal and external software quality attributes using source code examples. This chapter empirically validates this classification by using complete software systems. It reports the results of two conducted case studies. The objective of the first case study is to validate the classification of the investigated refactoring methods by using three software projects developed by undergraduate students and three open source software projects. The objective of the second case study is to validate the classification of the investigated refactoring to patterns by using four open source software projects.

6.1. Case Study I: Validating Refactoring Methods Classification

This case study is concerned with validating the classification of the investigated refactoring methods, presented in Chapter 5, by using real software projects which consist of three projects developed by undergraduate students in their course project and three open source projects.

6.1.1. Software Systems Background

The software systems used in this case study are three course projects and three open source projects. The course projects (Project 01, Project 02, and Project 03) were developed by undergraduate students taking "*ICS 102 Introduction to Computing*" course offered by the Department of Information and Computer Science at King Fahd University of Petroleum and Minerals. The open source projects were downloaded from SourceForge.net [12]. These open source projects are: JLOC [6], J2Sharp [3], and JNFS (Java Net File Sender) [8]. Table 6-9 summarizes some main characteristics of the studied software projects used in this case study.

Table 6-9. The characteristics of studied software projects in case study I

	Project	Language	# of Classes	Lines of Code	Description
Course Projects	Project 01	Java	3	406	A program for managing the cars in a car rental agency
	Project 02	Java	2	299	A program for managing a computer software store
	Project 03	Java	2	334	A program for managing a computer software store
Open Source Projects	JLOC	Java	6	308	An application for counting comment, blank, and source code lines
	J2Sharp	Java	4	434	An application for converting Java code into C# code
	JNFS	Java	8	431	An application for sending a file from client to server via the internet

6.1.2. Data Collection

This section describes the methodology used to collect data from subject systems. The methodology includes the following steps:

1. Look for opportunity to apply a refactoring method in a system.

2. Collect the internal quality metrics (described in Chapter 4) for the system before applying the refactoring method. The *Understand for Java* metrics tool [13] and the *Metamata* metrics tool [9] were used to collect the internal quality metrics.

3. Perform the refactoring method with the help of *IntelliJ IDEA* tool [2] and *RefactorIT* tool [11] and then compile the source code to make sure that the system works properly as before.

4. Collect the internal quality metrics for the system after applying the refactoring method. The *Understand for Java* metrics tool [13] and the *Metamata* metrics tool [9] were used to collect the internal quality metrics.

5. Report the changes in the internal quality metrics for each class in the system.

6. Map the changes in the internal quality metrics to the external quality attributes as described in Chapter 4.

7. Repeat steps 1 to 6 for all the investigated refactoring methods till there is no opportunity to apply any of the investigated refactoring methods.

Once the data is collected, we can classify the investigated refactoring methods based on their effect on the internal and the external quality attributes. Table 6-10 shows the number of times that each refactoring method was applied on each software project used in this case study.

Table 6-10. Number of applied refactoring methods on each software project in case study I

Refactoring Method	Project 01	Project 02	Project 03	JLOC	J2Sharp	JNFS
Consolidate Conditional Expression	-	2	-	-	-	-
Encapsulate Field	-	-	1	-	-	-
Extract Class	1	-	-	-	-	-
Extract Method	6	1	4	1	1	2
Hide Method	1	1	1	1	1	-
Inline Class	-	-	-	-	-	1
Inline Method	1	1	-	-	1	2
Inline Temp	1	2	1	2	1	-
Remove Setting Method	2	-	1	1	-	-
Replace Assignment with Initialization	1	-	1	1	-	2
Replace Magic Number with Symbolic Constant	2	1	1	1	1	2
Reverse Conditional	-	1	-	-	1	-

6.1.3. Results from Course Project 01

This section presents the results of the Course Project 01 from this case study. Table 6-11 provides the measurement results for the whole classes before applying any refactoring method. Table 6-12 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-13 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-11. Measurement results before applying any refactoring method: Course Project 01

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Car.java	1	0	0	11	1	11	10	44	7
Customer.java	1	0	0	8	1	8	7	35	3
DemoCarAgency.java	1	0	2	5	8	5	5	327	10

Table 6-12. Measurement results for the affected classes before and after applying refactoring methods: Course Project 01

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	Before	DemoCarAgency.java	1	0	2	5	8	5	5	327	10
	After	DemoCarAgency.java	1	0	2	6	8	6	6	336	15
Extract Method	Before	DemoCarAgency.java	1	0	2	6	8	6	6	336	15
	After	DemoCarAgency.java	1	0	2	7	8	7	7	347	21
Extract Method	Before	DemoCarAgency.java	1	0	2	7	8	7	7	347	21
	After	DemoCarAgency.java	1	0	2	8	8	8	8	353	28
Extract Method	Before	DemoCarAgency.java	1	0	2	8	8	8	8	353	28
	After	DemoCarAgency.java	1	0	2	9	8	9	9	359	36
Extract Method	Before	DemoCarAgency.java	1	0	2	9	8	9	9	359	36
	After	DemoCarAgency.java	1	0	2	10	8	10	10	370	45
Extract Method	Before	DemoCarAgency.java	1	0	2	10	8	10	10	370	45
	After	DemoCarAgency.java	1	0	2	11	8	11	11	382	55
Hide Method	Before	DemoCarAgency.java	1	0	2	11	8	11	11	382	55
	After	DemoCarAgency.java	1	0	2	11	8	11	11	382	55

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Magic Number with Symbolic Constant	Before	DemoCarAgency.java	1	0	2	11	8	11	11	382	55
	After	DemoCarAgency.java	1	0	2	11	8	11	11	383	55
Replace Magic Number with Symbolic Constant	Before	DemoCarAgency.java	1	0	2	11	8	11	11	383	55
	After	DemoCarAgency.java	1	0	2	11	8	11	11	384	55
Remove Setting Method	Before	Car.java	1	0	0	11	1	11	10	44	7
	After	Car.java	1	0	0	10	1	10	9	41	2
Remove Setting Method	Before	Customer.java	1	0	0	8	1	8	7	35	3
	After	Customer.java	1	0	0	7	1	7	6	32	1
Extract Class	Before	DemoCarAgency.java	1	0	2	11	8	11	11	384	55
		ShowMenu.java	-	-	-	-	-	-	-	-	-
	After	DemoCarAgency.java	1	0	3	8	9	8	8	317	28
		ShowMenu.java	1	0	0	3	2	3	3	73	3
Replace Assignment with Initialization	Before	DemoCarAgency.java	1	0	3	8	9	8	8	317	28
	After	DemoCarAgency.java	1	0	3	8	9	8	8	290	28
Inline Temp	Before	DemoCarAgency.java	1	0	3	8	9	8	8	290	28
	After	DemoCarAgency.java	1	0	3	8	9	8	8	289	28

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Inline Method	Before	DemoCarAgency.java	1	0	3	8	9	8	8	289	28
	After	DemoCarAgency.java	1	0	3	7	9	7	7	286	21

Table 6-13. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 01

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Class	-	-	↑	↓	↑	↓	↓	↓	↓
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

6.1.4. Results from Course Project 02

This section presents the results of the Course Project 02 from this case study. Table 6-14 provides the measurement results for the whole classes before applying any refactoring method. Table 6-15 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-16 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-14. Measurement results before applying any refactoring method: Course Project 02

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Project.java	1	0	0	6	1	6	5	30	2
TestProject.java	1	0	1	8	7	8	7	267	0

Table 6-15. Measurement results for the affected classes before and after applying refactoring methods: Course Project 02

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Hide Method	Before	TestProject.java	1	0	1	8	7	8	7	267	0
	After	TestProject.java	1	0	1	8	7	8	7	267	0
Inline Temp	Before	TestProject.java	1	0	1	8	7	8	7	267	0
	After	TestProject.java	1	0	1	8	7	8	7	262	0
Extract Method	Before	TestProject.java	1	0	1	8	7	8	7	262	0
	After	TestProject.java	1	0	1	9	7	9	8	265	6
Consolidate Conditional Expression	Before	TestProject.java	1	0	1	9	7	9	8	265	6
	After	TestProject.java	1	0	1	10	7	10	9	264	15
Consolidate Conditional Expression	Before	TestProject.java	1	0	1	10	7	10	9	264	15
	After	TestProject.java	1	0	1	11	7	11	10	263	25
Replace Magic Number with Symbolic Constant	Before	TestProject.java	1	0	1	11	7	11	10	263	25
	After	TestProject.java	1	0	1	11	7	11	10	264	25
Reverse Conditional	Before	TestProject.java	1	0	1	11	7	11	10	264	25
	After	TestProject.java	1	0	1	11	7	11	10	264	25

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Inline Method	Before	TestProject.java	1	0	1	11	7	11	10	264	25
	After	TestProject.java	1	0	1	10	7	10	9	261	15
Inline Temp	Before	TestProject.java	1	0	1	10	7	10	9	261	15
	After	TestProject.java	1	0	1	10	7	10	9	260	15

Table 6-16. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 02

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Consolidate Conditional Expression	-	-	-	↑	-	↑	↑	↓	↑
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-
Reverse Conditional	-	-	-	-	-	-	-	-	-

6.1.5. Results from Course Project 03

This section presents the results of the Course Project 03 from this case study. Table 6-17 provides the measurement results for the whole classes before applying any refactoring method. Table 6-18 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-19 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-17. Measurement results before applying any refactoring method: Course Project 03

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Store.java	1	0	0	13	1	13	12	62	16
StoreManager.java	1	0	1	11	4	11	11	272	0

Table 6-18. Measurement results for the affected classes before and after applying refactoring methods: Course Project 03

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field	Before	Store.java	1	0	0	13	1	13	12	62	16
	After	Store.java	1	0	0	15	1	15	14	68	35
Remove Setting Method	Before	Store.java	1	0	0	15	1	15	14	68	35
	After	Store.java	1	0	0	13	1	13	12	62	24
Hide Method	Before	StoreManager.java	1	0	1	11	4	11	11	272	0
	After	StoreManager.java	1	0	1	11	4	11	11	272	0
Replace Assignment with Initialization	Before	StoreManager.java	1	0	1	11	4	11	11	272	0
	After	StoreManager.java	1	0	1	11	4	11	11	267	0
Inline Temp	Before	StoreManager.java	1	0	1	11	4	11	11	267	0
	After	StoreManager.java	1	0	1	11	4	11	11	263	0
Extract Method	Before	StoreManager.java	1	0	1	11	4	11	11	263	0
	After	StoreManager.java	1	0	1	12	4	12	12	266	10
Extract Method	Before	StoreManager.java	1	0	1	12	4	12	12	266	10
	After	StoreManager.java	1	0	1	13	4	13	13	269	22

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	Before	StoreManager.java	1	0	1	13	4	13	13	269	22
	After	StoreManager.java	1	0	1	14	4	14	14	272	35
Extract Method	Before	StoreManager.java	1	0	1	14	4	14	14	272	35
	After	StoreManager.java	1	0	1	15	4	15	15	275	49
Replace Magic Number with Symbolic Constant	Before	Store.java	1	0	0	13	1	13	12	62	24
	After	Store.java	1	0	0	13	1	13	12	63	24

Table 6-19. Changes in the internal quality metrics caused by applying refactoring methods: Course Project 03

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field	-	-	-	↑	-	↑	↑	↑	↑
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Inline Temp	-	-	-	-	-	-	-	↓	-
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

6.1.6. Results from JLOC

This section presents the results of the JLOC project from this case study. Table 6-20 provides the measurement results for the whole classes before applying any refactoring method. Table 6-21 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-22 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-20. Measurement results before applying any refactoring method: JLOC

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
BasicFileInfo.java	1	0	0	10	1	10	10	39	35
CommonCounter.java	1	0	1	2	5	2	2	53	1
Gui.java	1	0	1	3	8	3	3	77	1
ILineCounter.java	1	1	1	1	2	1	1	7	0
Main.java	1	0	2	1	2	1	1	39	0
Table.java	2	0	3	17	5	11	11	91	8

Table 6-21. Measurement results for the affected classes before and after applying refactoring methods: JLOC

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Assignment with Initialization	Before	Gui.java	1	0	1	3	8	3	3	77	1
	After	Gui.java	1	0	1	3	8	3	3	74	1
Remove Setting Method	Before	CommonCounter.java	1	0	1	2	5	2	2	53	1
	After	CommonCounter.java	1	0	1	1	5	1	1	50	0
Inline Temp	Before	Main.java	1	0	2	1	2	1	1	39	0
	After	Main.java	1	0	2	1	2	1	1	37	0
Extract Method	Before	Table.java	2	0	3	17	5	11	11	91	8
	After	Table.java	2	0	3	18	5	12	12	94	9
Inline Temp	Before	Table.java	2	0	3	18	5	12	12	94	9
	After	Table.java	2	0	3	18	5	12	12	93	9
Hide Method	Before	Table.java	2	0	3	18	5	12	12	93	9
	After	Table.java	2	0	3	18	5	12	12	93	9
Replace Magic Number with Symbolic Constant	Before	Gui.java	1	0	1	3	8	3	3	74	1
	After	Gui.java	1	0	1	3	8	3	3	76	1

Table 6-22. Changes in the internal quality metrics caused by applying refactoring methods: JLOC

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Inline Temp	-	-	-	-	-	-	-	↓	-
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

6.1.7. Results from J2Sharp

This section presents the results of the J2Sharp project from this case study. Table 6-23 provides the measurement results for the whole classes before applying any refactoring method. Table 6-24 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-25 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-23. Measurement results before applying any refactoring method: J2Sharp

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
J2Sharp.java	1	0	0	6	4	6	5	171	6
J2SharpGUI.java	2	0	2	7	12	7	6	206	5
Main.java	1	0	1	1	1	1	1	28	0
MyInternalFrame.java	2	0	0	2	2	2	0	29	0

Table 6-24. Measurement results for the affected classes before and after applying refactoring methods: J2Sharp

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Magic Number with Symbolic Constant	Before	MyInternalFrame.java	2	0	0	2	2	2	0	29	0
	After	MyInternalFrame.java	2	0	0	2	2	2	0	31	0
Extract Method	Before	Main.java	1	0	1	1	1	1	1	28	0
	After	Main.java	1	0	1	2	1	2	2	31	1
Inline Temp	Before	J2SharpGUI.java	2	0	2	7	12	7	6	206	5
	After	J2SharpGUI.java	2	0	2	7	12	7	6	205	5
Inline Method	Before	J2SharpGUI.java	2	0	2	7	12	7	6	205	5
	After	J2SharpGUI.java	2	0	2	6	12	6	5	202	4
Reverse Conditional	Before	J2SharpGUI.java	2	0	2	6	12	6	5	202	4
	After	J2SharpGUI.java	2	0	2	6	12	6	5	202	4
Hide Method	Before	J2Sharp.java	1	0	0	6	4	6	5	171	6
	After	J2Sharp.java	1	0	0	6	4	6	5	171	6

Table 6-25. Changes in the internal quality metrics caused by applying refactoring methods: J2Sharp

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-
Reverse Conditional	-	-	-	-	-	-	-	-	-

6.1.8. Results from JNFS

This section presents the results of the JNFS project from this case study. Table 6-26 provides the measurement results for the whole classes before applying any refactoring method. Table 6-27 provides the measurement results for the affected classes as a result of applying the appropriate refactoring methods. Table 6-28 presents the changes in the internal quality metrics caused by applying the appropriate refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-26. Measurement results before applying any refactoring method: JNFS

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Compteur.java	2	0	0	5	0	5	4	31	0
Fichier.java	1	0	0	4	2	4	3	23	3
JnfsClient.java	2	0	4	2	7	2	1	74	0
JnfsServeur.java	2	0	3	2	7	2	1	75	0
Outils.java	1	0	0	1	0	1	0	6	0
Parametres.java	1	0	0	2	0	2	1	10	0
MainFrm.java	2	0	2	13	9	13	12	143	58
ShowIp.java	2	0	0	5	6	5	4	69	6

Table 6-27. Measurement results for the affected classes before and after applying refactoring methods: JNFS

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Inline Class	Before	JnfsClient.java	2	0	4	2	7	2	1	74	0
		Parametres.java	1	0	0	2	0	2	1	10	0
	After	JnfsClient.java	2	0	3	3	6	3	2	78	1
		Parametres.java	-	-	-	-	-	-	-	-	-
Inline Method	Before	MainFrm.java	2	0	2	13	9	13	12	143	58
	After	MainFrm.java	2	0	2	12	9	12	11	139	47
Inline Method	Before	ShowIp.java	2	0	0	5	6	5	4	69	6
	After	ShowIp.java	2	0	0	4	6	4	3	66	3
Replace Assignment with Initialization	Before	MainFrm.java	2	0	2	12	9	12	11	139	47
	After	MainFrm.java	2	0	2	12	9	12	11	136	47
Replace Assignment with Initialization	Before	ShowIp.java	2	0	0	4	6	4	3	66	3
	After	ShowIp.java	2	0	0	4	6	4	3	64	3
Extract Method	Before	JnfsClient.java	2	0	3	3	6	3	2	78	0
	After	JnfsClient.java	2	0	3	4	6	4	3	82	1

Refactoring Method		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	Before	JnfsServeur.java	2	0	3	2	7	2	1	75	0
	After	JnfsServeur.java	2	0	3	3	7	3	2	79	1
Replace Magic Number with Symbolic Constant	Before	JnfsClient.java	2	0	3	4	6	4	3	82	1
	After	JnfsClient.java	2	0	3	4	6	4	3	83	1
Replace Magic Number with Symbolic Constant	Before	JnfsServeur.java	2	0	3	3	7	3	2	79	1
	After	JnfsServeur.java	2	0	3	3	7	3	2	80	1

Table 6-28. Changes in the internal quality metrics caused by applying refactoring methods: JNFS

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Inline Class	-	-	↓	↑	↓	↑	↑	↑	↑
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

6.1.9. Discussion of Results

From the results presented in Table 6-13, Table 6-16, Table 6-19, Table 6-22, Table 6-25, and Table 6-28 that show the changes in the internal quality metrics of the studied software projects, we can classify the investigated refactoring methods based on their effect on the internal quality metrics. Then, we can classify the investigated refactoring methods according to the external quality attributes by mapping the changes in the internal quality metrics to the external quality attributes as described in Chapter 4.

Table 6-29 presents the classification of the investigated refactoring methods based on the internal quality metrics using empirical results from the studied software projects, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-30 presents the classification of the investigated refactoring methods based on the external quality attributes using empirical results from the studied software projects, where "↑" symbol represents an increase (improve) in external quality attribute except for the testability (testing effort) it means an impair, "↓" symbol represents a decrease (impair) in external quality attribute except for the testability (testing effort) it means an improve, and "-" symbol represents no change in external quality attribute.

Table 6-29. Classification of refactoring methods based on internal software quality attributes using empirical results

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field	-	-	-	↑	-	↑	↑	↑	↑
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Hide Method	-	-	-	-	-	-	-	-	-
Reverse Conditional	-	-	-	-	-	-	-	-	-
Inline Method	-	-	-	↓	-	↓	↓	↓	↓
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Consolidate Conditional Expression	-	-	-	↑	-	↑	↑	↓	↑
Extract Class	-	-	↑	↓	↑	↓	↓	↓	↓
Inline Class	-	-	↓	↑	↓	↑	↑	↑	↑
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

Table 6-30. Classification of refactoring methods based on external software quality attributes using empirical results

Refactoring Method	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability	Reliability
Inline Method	↓	↓	↓	↓	↓	↓	↑
Remove Setting Method	↓	↓	↓	↓	↓	↓	↑
Inline Temp	↓	↓	↓	↓	↓	↓	↑
Replace Assignment with Initialization	↓	↓	↓	↓	↓	↓	↑
Extract Class	↓	↓	↓	↓	↓	↓	↑
Encapsulate Field	↑	↑	↑	↑	↑	↑	↓
Extract Method	↑	↑	↑	↑	↑	↑	↓
Inline Class	↑	↑	↑	↑	↑	↑	↓
Replace Magic Number with Symbolic Constant	↑	↑	↑	↑	↑	↑	↓
Hide Method	-	-	-	-	-	-	-
Reverse Conditional	-	-	-	-	-	-	-
Consolidate Conditional Expression	↓	↓	↓	↓	↓	↑	↓

This case study provided a number of interesting results which can be observed as follows (see Chapter 5 for further analysis on the effect of the investigated refactoring methods on the internal quality metrics):

- The empirical results presented in Table 6-29 and Table 6-30 which are provided by the course projects and the open source projects, validate the classification of the investigated refactoring methods presented in Chapter 5.

- The empirical results obtained from software projects developed by undergraduate students are same as the empirical results obtained from open source software projects. This strengthens our refactoring classification as we consider different programming abilities (students / beginners and professionals).

- During the analysis of the empirical results presented in Table 6-12, Table 6-15, Table 6-18, Table 6-21, Table 6-24, and Table 6-27, we observed that all of the investigated refactoring methods are dealing with one class except “Extract Class” and “Inline Class” are dealing with two classes. This is because that “Extract Class” creates (extracts) new class from old class, while “Inline Class” puts (inline) a class into another class.

- The empirical results presented in Table 6-12 and Table 6-27 support the earlier explanation in Chapter 5 that “Extract Class” increases the number of classes in the system, whereas “Inline Class” reduces the number of classes in the system.

- We can observe from Table 6-29 that “Encapsulate Field”, “Extract Method”, and “Inline Class” increase the class size in terms of WMC, NOM, and LOC metrics since they introduce new methods. On the other hand, “Inline Method”, “Remove Setting Method”, and “Extract Class” reduce the class size in terms of WMC, NOM, and LOC metrics.

- We can observe from Table 6-29 and Table 6-30 that the refactoring methods that are inverse to each other e.g. (“Extract Method” and “Inline Method”), and (“Extract Class” and “Inline Class”) have inverse effect on the internal and the external quality attributes.

- One of the objectives of good software design is to reduce the coupling where possible and increase the cohesion where possible [40]. We can observe from Table 6-29 that “Inline Class” reduces the coupling (CBO and FOUT), while “Extract Class” increases the coupling (CBO and FOUT). Additionally, “Inline Method”, “Remove Setting Method”, and “Extract Class” increase the cohesion (reduce LCOM) while “Encapsulate Field”, “Extract Method”, “Consolidate Conditional Expression”, and “Inline Class” reduce the cohesion (increase LCOM).

Another interesting observation obtained from Table 6-29 is that the refactoring methods that are not inverse to each other e.g. (“Encapsulate Field” and “Remove Setting Method”), (“Extract Method” and “Remove Setting Method”), (“Inline Temp” and

"Replace Magic Number with Symbolic Constant"), and ("Replace Assignment with Initialization" and "Replace Magic Number with Symbolic Constant") have inverse (conflict) effect on the all internal quality metrics. Table 6-31 presents the refactoring methods that are not inverse to each other, but they have inverse effect on the all internal quality metrics.

This led us to investigate this situation where some refactoring methods are not inverse to each other, but they have inverse effect on the all internal quality metrics. Therefore, we need to apply these refactoring methods together and then observe the changes in the internal quality metrics for the whole system. This, in turn, helps us to understand which one of the refactoring methods is more dominant (has more effect on the internal quality metrics). We composed four groups of refactoring methods. Each group consists of two refactoring methods that have inverse effect on the all internal quality metrics.

Table 6-31. Refactoring methods that have inverse effect on the internal quality metrics

Refactoring Method	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field	-	-	-	↑	-	↑	↑	↑	↑
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Extract Method	-	-	-	↑	-	↑	↑	↑	↑
Remove Setting Method	-	-	-	↓	-	↓	↓	↓	↓
Inline Temp	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-
Replace Assignment with Initialization	-	-	-	-	-	-	-	↓	-
Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	↑	-

To calculate the internal quality metrics for the whole system, we calculated the internal quality metrics for all classes in that system and then we aggregated the internal quality metrics values by taking their sum and their average. Table 6-32 and Table 6-33 provide the aggregated measurement results by taking their sum and their average respectively. We can see from Table 6-32 and Table 6-33 that two types of the aggregation provide the same results.

Table 6-32. Sum of measurement results before and after applying group of refactoring methods

Group of Refactoring Methods		Software Project	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field & Remove Setting Method	Before	Course Project 03	2	0	1	24	5	24	23	334	16
	After	Course Project 03	2	0	1	25	5	25	24	337	28
Extract Method & Remove Setting Method	Before	Course Project 03	2	0	1	26	5	26	25	340	35
	After	Course Project 03	2	0	1	26	5	26	25	340	35
Extract Method & Remove Setting Method	Before	JLOC	7	1	8	34	23	28	28	303	45
	After	JLOC	7	1	8	34	23	28	28	303	45
Inline Temp & Replace Magic Number with Symbolic Constant	Before	JLOC	7	1	8	34	23	28	28	301	45
	After	JLOC	7	1	8	34	23	28	28	301	45
Inline Temp & Replace Magic Number with Symbolic Constant	Before	J2Sharp	6	0	3	16	19	16	12	434	11
	After	J2Sharp	6	0	3	16	19	16	12	434	11
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	Before	Course Project 03	2	0	1	24	5	24	23	334	24
	After	Course Project 03	2	0	1	24	5	24	23	334	24
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	Before	JNFS	12	0	8	31	30	31	24	418	53
	After	JNFS	12	0	8	31	30	31	24	418	53

Table 6-33. Average of measurement results before and after applying group of refactoring methods

Group of Refactoring Methods		Software Project	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field & Remove Setting Method	Before	Course Project 03	1	0	0.5	12	2.5	12	11.5	167	8
	After	Course Project 03	1	0	0.5	12.5	2.5	12.5	12	168.5	14
Extract Method & Remove Setting Method	Before	Course Project 03	1	0	0.5	13	2.5	13	12.5	170	17.5
	After	Course Project 03	1	0	0.5	13	2.5	13	12.5	170	17.5
Extract Method & Remove Setting Method	Before	JLOC	1.16	0.16	1.33	5.66	3.83	4.66	4.66	50.5	7.5
	After	JLOC	1.16	0.16	1.33	5.66	3.83	4.66	4.66	50.5	7.5
Inline Temp & Replace Magic Number with Symbolic Constant	Before	JLOC	1.16	0.16	1.33	5.66	3.83	4.66	4.66	50.17	7.5
	After	JLOC	1.16	0.16	1.33	5.66	3.83	4.66	4.66	50.17	7.5
Inline Temp & Replace Magic Number with Symbolic Constant	Before	J2Sharp	1.5	0	0.75	4	4.75	4	3	108.5	2.75
	After	J2Sharp	1.5	0	0.75	4	4.75	4	3	108.5	2.75
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	Before	Course Project 03	1	0	0.5	12	2.5	12	11.5	167	12
	After	Course Project 03	1	0	0.5	12	2.5	12	11.5	167	12
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	Before	JNFS	1.71	0	1.14	4.42	4.28	4.42	3.42	59.71	7.57
	After	JNFS	1.71	0	1.14	4.42	4.28	4.42	3.42	59.71	7.57

Table 6-34 presents the changes in the internal quality metrics caused by applying group of refactoring methods, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-34. Changes in the internal quality metrics caused by applying group of refactoring methods

Group of Refactoring Methods	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Encapsulate Field & Remove Setting Method	-	-	-	↑	-	↑	↑	↑	↑
Extract Method & Remove Setting Method	-	-	-	-	-	-	-	-	-
Inline Temp & Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	-	-
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-	-	-

The first group ("Encapsulate Field" and "Remove Setting Method") increases RFC, WMC, NOM, LOC, and LCOM metrics value. This is because that "Encapsulate Field" provides (two methods) setter and getter methods for a field, while "Remove Setting Method" removes a setting method for a field. Therefore, there is one extra method after applying this group of refactoring which is the getter method. This, in turn,

increases the number of methods in the system, increases the source code statements, and makes the system less cohesive.

The second group ("Extract Method" and "Remove Setting Method") does not affect any of the investigated internal quality metrics. This is because that "Extract Method" creates (extracts) new method, while "Remove Setting Method" removes a setting method. Therefore, after applying this group of refactoring, the internal quality metrics including the number of methods of the system will not change.

The third group ("Inline Temp" and "Replace Magic Number with Symbolic Constant") does not affect any of the investigated internal quality metrics. This is because that "Inline Temp" replaces all references to the temporary variable with an expression and removes the temporary variable, while "Replace Magic Number with Symbolic Constant" creates a constant and replaces the number with it. Therefore, "Inline Temp" reduces the source code statements by one line, whereas "Replace Magic Number with Symbolic Constant" increases the source code statements by one line. After applying this group of refactoring, the internal quality metrics including the number of source code statements of the system will not change.

The fourth group ("Replace Assignment with Initialization" and "Replace Magic Number with Symbolic Constant") does not affect any of the investigated internal quality metrics. This is because that "Replace Assignment with Initialization" makes direct

initialization of variable instead of declare the variable and then assign a value to it, while "Replace Magic Number with Symbolic Constant" creates a constant and replaces the number with it. Therefore, "Replace Assignment with Initialization" reduces the source code statements by one line, whereas "Replace Magic Number with Symbolic Constant" increases the source code statements by one line. After applying this group of refactoring, the internal quality metrics including the number of source code statements of the system will not change.

To study the effect of the refactoring groups, discussed above, on the external quality attributes, we map the changes in the internal quality metrics presented in Table 6-34 to the external quality attributes as described in Chapter 4. Table 6-35 presents the effect of the refactoring groups on the external quality attributes, where "↑" symbol represents an increase (improve) in external quality attribute except for the testability (testing effort) it means an impair, "↓" symbol represents a decrease (impair) in external quality attribute except for the testability (testing effort) it means an improve, and "-" symbol represents no change in external quality attribute.

Table 6-35. The effect of refactoring groups on the external quality attributes

Groups of Refactoring Methods	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability	Reliability
Encapsulate Field & Remove Setting Method	↑	↑	↑	↑	↑	↑	↓
Extract Method & Remove Setting Method	-	-	-	-	-	-	-
Inline Temp & Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-
Replace Assignment with Initialization & Replace Magic Number with Symbolic Constant	-	-	-	-	-	-	-

6.2. Case Study II: Validating Refactoring to Patterns Classification

This case study is concerned with validating the classification of the investigated refactoring to patterns, presented in Chapter 5, by using real software projects which consist of four open source projects.

6.2.1. Software Systems Background

The software systems used in this case study are four open source projects. The open source projects were downloaded from SourceForge.net [12]. These open source projects are: HTML Parser [1], Java Class Browser [4], Java Neural Network Trainer [5], and JMK (Make in Java) [7]. Table 6-36 summarizes some main characteristics of these projects.

Table 6-36. The characteristics of studied software projects in case study II

Project	Language	# of Classes	Lines of Code	Description
HTML Parser	Java	202	25992	An application used to parse HTML in either a linear or nested fashion. Primarily used for transformation or extraction
Java Class Browser	Java	9	647	An application used to view the class file names in a Java archive or Directory with multiple archives
Java Neural Network Trainer	Java	11	1171	An application used to add new training algorithms and training patterns
JMK	Java	47	5016	An application used to ensure that a set of files is in a consistent state

6.2.2. Data Collection

The data was collected from subject systems as described in Section 6.1.2. The only difference is that we are looking for refactoring to patterns opportunities instead of refactoring method opportunities.

6.2.3. Results from HTML Parser

This section presents the results of the HTML Parser from this case study. Table 6-37 and Table 6-38 provide the measurement results for the "lexer" package and "tests" package respectively before applying any refactoring to patterns. Table 6-39 provides the measurement results for the affected classes as a result of applying the appropriate refactoring to patterns.

Table 6-37. Measurement results for lexer package: HTML Parser

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Cursor.java	1	0	1	9	3	9	8	76	14
Lexer.java	1	0	9	35	9	35	31	773	377
Page.java	1	1	7	31	15	31	27	522	147
PageIndex.java	1	0	4	24	2	24	21	171	102
Source.java	2	0	1	21	8	21	18	248	0
Stream.java	2	0	0	10	1	10	8	158	2

Table 6-38. Measurement results for tests package: HTML Parser

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
AllTests.java	1	0	2	2	1	1	1	35	0
AssertXmlEqualsTest.java	2	0	0	41	2	11	10	44	45
BadTagIdentifier.java	1	0	5	3	3	3	2	33	1
FunctionalTests.java	2	0	6	10	3	9	5	82	10
InstanceofPerformance Test.java	1	0	7	11	4	14	6	83	11
LineNumberAssigned ByNodeReaderTest.java	2	0	3	42	1	12	11	92	55
ParserTest.java	2	0	11	42	9	45	21	705	210
ParserTestCase.java	1	58	15	30	8	30	29	454	354
PerformanceTest.java	1	0	5	9	2	12	3	85	1

Table 6-39. Measurement results for the affected classes before and after applying refactoring to patterns: HTML Parser

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Constructors with Creation Methods	Before	Page.java	1	1	7	31	15	31	27	522	147
		Source.java	2	0	1	21	8	21	18	248	0
	After	Page.java	1	1	7	31	15	31	27	522	147
		Source.java	2	0	1	23	8	23	20	254	32
Form Template Method	Before	ParserTestCase.java	2	58	15	30	8	30	29	454	354
		AssertXmlEqualsTest.java	3	0	0	41	2	11	10	44	45
		LineNumberAssignedByNodeReaderTest.java	3	0	3	42	1	12	11	92	55
	After	ParserTestCase.java	2	58	15	32	8	32	31	458	413
		AssertXmlEqualsTest.java	3	0	0	43	2	11	10	44	45
		LineNumberAssignedByNodeReaderTest.java	3	0	3	44	1	12	11	92	55

6.2.4. Results from Java Class Browser

This section presents the results of the Java Class Browser from this case study. Table 6-40 provides the measurement results for the whole classes before applying any refactoring to patterns. Table 6-41 provides the measurement results for the affected classes as a result of applying the appropriate refactoring to patterns.

Table 6-40. Measurement results before applying any refactoring to patterns: Java Class Browser

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
ArchiveFilter.java	2	0	0	5	2	5	5	31	8
ClassBrowser.java	2	0	4	14	19	14	13	155	78
ClassFileTableModel.java	2	0	0	6	3	6	5	58	4
ClassInfo.java	1	0	0	0	0	0	0	3	0
MyFilter.java	1	0	0	1	2	1	1	10	0
ReadJar.java	1	0	1	5	6	5	5	101	10
ShowClasses.java	1	0	2	1	8	1	1	44	0
TableMap.java	2	1	0	10	3	10	10	38	0
TableSorter.java	3	0	14	27	6	17	15	207	59

Table 6-41. Measurement results for the affected classes before and after applying refactoring to patterns: Java Class Browser

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Replace Conditional Dispatcher with Command	Before	FifthCase.java	-	-	-	-	-	-	-	-	-
		FirstCase.java	-	-	-	-	-	-	-	-	-
		FourthCase.java	-	-	-	-	-	-	-	-	-
		Handler.java	-	-	-	-	-	-	-	-	-
		SecondCase.java	-	-	-	-	-	-	-	-	-
		TableSorter.java	3	0	14	27	6	17	15	207	59
		ThirdCase.java	-	-	-	-	-	-	-	-	-
	After	FifthCase.java	2	0	0	3	3	2	1	24	0
		FirstCase.java	2	0	0	3	3	2	1	24	0
		FourthCase.java	2	0	0	3	3	2	1	23	0
		Handler.java	1	5	0	1	1	1	0	11	0
		SecondCase.java	2	0	0	3	3	2	1	23	0
		TableSorter.java	3	0	16	29	8	19	17	162	88
		ThirdCase.java	2	0	0	3	3	2	1	23	0

6.2.5. Results from Java Neural Network Trainer

This section presents the results of the Java Neural Network Trainer from this case study. Table 6-42 provides the measurement results for the whole classes before applying any refactoring to patterns. Table 6-43 provides the measurement results for the affected classes as a result of applying the appropriate refactoring to patterns.

Table 6-42. Measurement results before applying any refactoring to patterns: Java Neural Network Trainer

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
BackProp.java	3	0	1	18	1	9	7	70	3
GA.java	3	0	1	21	2	12	11	167	45
NeuralNetwork.java	1	0	0	20	2	20	19	155	39
Pso.java	3	0	1	17	1	8	7	104	7
QuickProp.java	3	0	1	17	1	8	6	130	11
Trainer.java	2	4	2	9	4	9	8	62	12
TrainerListener.java	1	2	2	3	2	3	3	7	3
TestNN.java	5	0	22	19	12	18	6	314	7
Problem.java	1	2	2	8	2	8	7	39	21
RealNumbers.java	2	0	2	14	2	6	5	84	0
XOR.java	2	0	2	11	2	3	2	53	1

Table 6-43. Measurement results for the affected classes before and after applying refactoring to patterns: Java Neural Network Trainer

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Unify Interfaces	Before	Pso.java	3	0	1	17	1	8	7	104	7
		QuickProp.java	3	0	1	17	1	8	6	130	11
		Trainer.java	2	4	2	9	4	9	8	62	12
	After	Pso.java	3	0	1	19	1	8	7	104	7
		QuickProp.java	3	0	1	19	1	8	6	130	11
		Trainer.java	2	4	2	11	4	11	10	66	50
Compose Method	Before	QuickProp.java	3	0	1	19	1	8	6	130	11
	After	QuickProp.java	3	0	1	21	1	10	8	136	16
Introduce Null Object	Before	Trainer.java	2	4	2	11	4	11	10	66	50
		NullTrainer.java	-	-	-	-	-	-	-	-	-
		RealNumbers.java	2	0	2	14	2	6	5	84	0
	After	Trainer.java	2	5	2	11	4	11	10	66	50
		NullTrainer.java	3	0	0	14	1	1	1	7	0
		RealNumbers.java	2	0	2	14	2	6	5	82	0

6.2.6. Results from JMK

This section presents the results of the JMK from this case study. Table 6-44 provides the measurement results for the "jmk" package before applying any refactoring to patterns. Table 6-45 provides the measurement results for the affected classes as a result of applying the appropriate refactoring to patterns.

Table 6-44. Measurement results for jmk package: JMK

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Expression.java	1	17	4	1	4	0	0	7	0
BinaryFileOperator.java	1	0	9	8	4	4	2	41	3
FileOperator.java	1	0	10	29	7	12	5	133	30
Make.java	1	0	21	52	13	55	24	303	309
FunctionCastException.java	3	0	1	0	1	0	0	9	0
GlobalTable.java	1	0	21	32	5	3	2	411	3
ClassOperator.java	1	0	12	8	4	6	3	57	3
Loader.java	1	0	28	32	27	32	29	1118	131
CommandFailedException.java	3	0	1	0	1	0	0	9	0
Global.java	2	0	5	5	5	0	0	30	0
CreateOperator.java	1	0	12	8	4	6	3	50	3
Matcher.java	1	0	2	13	2	0	0	62	0
Environment.java	1	0	2	1	2	0	0	16	0
Command.java	1	0	14	14	11	0	0	67	0
Function.java	1	0	3	2	3	0	2	8	1
CvsInfo.java	1	0	4	13	2	13	5	52	10

Class Name	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
ExecOperator.java	1	0	16	17	4	7	3	74	3
NoteOperator.java	1	0	3	2	3	2	2	11	1
ParseError.java	3	0	2	3	2	2	2	20	1
Operator.java	1	0	4	2	4	0	2	8	1
Pattern.java	1	0	5	5	5	0	0	26	0
ReverseFunction.java	1	0	3	5	3	5	2	32	1
ReverseOperator.java	1	0	4	4	4	4	2	21	1
Rule.java	1	0	10	34	8	22	10	217	91
StringList.java	1	0	2	8	2	14	7	66	6
StringListCastException.java	3	0	1	0	1	0	0	9	0
StringUtils.java	1	0	9	26	4	2	2	184	53
Value.java	1	1	0	0	0	0	0	5	0
WildCardFilter.java	1	0	5	12	3	11	3	98	3
WrongArgCountException.java	3	0	1	0	1	0	0	9	0

Table 6-45. Measurement results for the affected classes before and after applying refactoring to patterns: JMK

Refactoring to Patterns		Class Name	Inheritance		Coupling			Size			Cohesion
			DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Chain Constructors	Before	Loader.java	1	0	28	32	27	32	29	1118	131
	After	Loader.java	1	0	28	32	27	32	29	1113	131

6.2.7. Discussion of Results

From the measurement results presented in Table 6-39, Table 6-41, Table 6-43, and Table 6-45, we can classify the investigated refactoring to patterns based on their effect on the internal quality metrics. Then, we can classify the investigated refactoring to patterns according to the external quality attributes by mapping the changes in the internal quality metrics to the external quality attributes as described in Chapter 4.

Table 6-46 presents the classification of the investigated refactoring to patterns based on the internal quality metrics using empirical results from the studied software projects, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

Table 6-47 presents the classification of the investigated refactoring to patterns based on the external quality attributes using empirical results from the studied software projects, where "↑" symbol represents an increase (improve) in external quality attribute except for the testability (testing effort) it means an impair, "↓" symbol represents a decrease (impair) in external quality attribute except for the testability (testing effort) it means an improve, and "-" symbol represents no change in external quality attribute.

Table 6-46. Classification of refactoring to patterns based on internal software quality attributes using empirical results

Refactoring to Patterns	Inheritance		Coupling			Size			Cohesion
	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Compose Method	-	-	-	↑	-	↑	↑	↑	↑
Form Template Method	-	-	-	↑	-	↑	↑	↑	↑
Replace Constructors with Creation Methods	-	-	-	↑	-	↑	↑	↑	↑
Unify Interfaces	-	-	-	↑	-	↑	↑	↑	↑
Chain Constructors	-	-	-	-	-	-	-	↓	-
Introduce Null Object	-	↑	-	-	-	-	-	↓	-
Replace Conditional Dispatcher with Command	-	-	↑	↑	↑	↑	↑	↓	↑

Table 6-47. Classification of refactoring to patterns based on external software quality attributes using empirical results

Refactoring to Patterns	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability	Reliability
Compose Method	↑	↑	↑	↑	↑	↑	↓
Form Template Method	↑	↑	↑	↑	↑	↑	↓
Replace Constructors with Creation Methods	↑	↑	↑	↑	↑	↑	↓
Unify Interfaces	↑	↑	↑	↑	↑	↑	↓
Chain Constructors	↓	↓	↓	↓	↓	↓	↑
Introduce Null Object	↓	↓	↓	↓	↓	↓	↑
Replace Conditional Dispatcher with Command	↓	↓	↓	↓	↓	↑	↓

This case study provided a number of interesting results which can be observed as follows (see Chapter 5 for further analysis on the effect of the investigated refactoring to patterns on the internal quality metrics):

- The empirical results presented in Table 6-46 and Table 6-47 which are provided by the open source software projects, validate the classification of the investigated refactoring to patterns presented in Chapter 5.

- During the analysis of the empirical results presented in Table 6-39, Table 6-41, Table 6-43, and Table 6-45, we observed that all of the investigated refactoring to patterns are dealing with more than one class except "Chain Constructors" and "Compose Method" which are dealing with one class.

- The empirical results presented in Table 6-41 and Table 6-43 support the earlier explanation in Chapter 5 that "Introduce Null Object" and "Replace Conditional Dispatcher with Command" increase the number of classes in the system.

- We can observe from Table 6-46 that "Compose Method", "Form Template Method", "Replace Constructors with Creation Methods", and "Unify Interfaces" increase the class size in terms of WMC, NOM, and LOC metrics as they introduce new methods.

- One of the objectives of good software design is to reduce the coupling where possible and increase the cohesion where possible [40]. We can observe from

Table 6-46 that “Replace Conditional Dispatcher with Command” increases the coupling (CBO and FOUT). Additionally, “Compose Method”, “Form Template Method”, “Replace Constructors with Creation Methods”, “Unify Interfaces”, and “Replace Conditional Dispatcher with Command” reduce the cohesion (increase LCOM). Therefore, none of the investigated refactoring to patterns can help to achieve this objective.

Chapter 7

Conclusion

This chapter concludes the thesis with an overview of main contributions and directions for future work.

7.1. Major Contributions

This thesis has classified set of refactorings based on internal and external software quality attributes. In particular the following contributions have been made:

1. A set of object-oriented metrics that affect external software quality attributes was identified based on available literature.

2. The Impact of refactoring methods and refactoring to patterns on internal software quality metrics (inheritance, coupling, size, and cohesion) was studied by observing the changes in the internal quality metrics caused by applying refactorings.

3. The Impact of refactoring methods and refactoring to patterns on external software quality attributes was studied by relating internal software quality metrics to external software quality attributes.

4. A classification of refactoring methods based on their measurable effect on internal and external software quality attributes was proposed.

5. A classification of refactoring to patterns based on their measurable effect on internal and external software quality attributes was proposed.

6. An empirical validation was conducted to validate the refactorings classification in the context of real software projects.

7.2. Future Work

Additional research directions that need to be explored in future work include the following:

- Investigate the effect of refactoring methods and refactoring to patterns on different set of internal software quality metrics such as metrics suites proposed by Briand et al. [19, 20] and MOOD metrics suite [32]. Then use these metrics to classify the refactorings.

- It is also interesting to investigate the effect of refactoring methods and refactoring to patterns on other external software quality attributes such as

performance, correctness, and portability. Then use these attributes to classify the refactorings.

- Use our approach to classify a more extended set of refactoring methods and refactoring to patterns based on software quality attributes to form a large classification catalog.

- Additional empirical validations and case studies are also needed to further support the findings of this research.

- Some refactoring methods and refactoring to patterns have inverse (conflict) effect on software quality attributes. Therefore, another area of research is to form large groups of refactorings that have conflict effect on software quality attributes and then study their impact on software quality attributes.

- Some refactoring methods and refactoring to patterns such as “Extract Class”, “Introduce Null Object”, and “Replace Conditional Dispatcher with Command” introduce new classes in the system which can be considered as side effect on the overall system quality. Our work focused on studying the effect of these refactorings on class level. Therefore, it is interesting to study the side effect of these kinds of refactorings on the overall system quality.

References

[1] HTML Parser, http://sourceforge.net/projects/htmlparser/, (Accessed: 1 January 2008).

[2] IntelliJ IDEA, http://www.jetbrains.com/idea/, (Accessed: 1 January 2008).

[3] J2Sharp, http://sourceforge.net/projects/j2sharp/, (Accessed: 1 January 2008).

[4] Java Class Browser, http://sourceforge.net/projects/classbrowser/, (Accessed: 1 January 2008).

[5] Java Neural Network Trainer, http://sourceforge.net/projects/javanntrain/, (Accessed: 1 January 2008).

[6] JLOC, http://sourceforge.net/projects/jloc/, (Accessed: 1 January 2008).

[7] JMK, http://sourceforge.net/projects/jmk/, (Accessed: 1 January 2008).

[8] JNFS, http://sourceforge.net/projects/jnfs/, (Accessed: 1 January 2008).

[9] MetaMata, www.metamata.com,

[10] Refactoring Website, http://www.refactoring.com/, (Accessed: 1 January 2008).

[11] RefactorIT, http://www.aqris.com/display/ap/RefactorIt/, (Accessed: 1 January 2008).

[12] SourceForge.net, http://sourceforge.net/index.php, (Accessed: 1 January 2008).

[13] Understand for Java, http://www.scitools.com/uj.html, (Accessed: 1 January 2008).

[14] V. Basili, L. Briand, W. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering 22 (10) (1996) 751-761.

[15] B. Bois, S. Demeyer, J. Verelst, Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?, Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), 2005, pp. 334-343.

[16] B. Bois, S. Demeyer, J. Verelst, Refactoring - Improving Coupling and Cohesion of Existing Code, Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04), 2004, pp. 144-151.

[17] B. Bois, T. Mens, Describing the impact of refactoring on internal program quality, Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications, 2003, pp. 37-48.

[18] F. Bravo, A Logic Meta-Programming Framework for Supporting the Refactoring Process, Master Thesis, Vrije Universiteit Brussel, 2003.

[19] L. Briand, J. Daly, J. Wust, A Unified Framework for Cohesion Measurement in Object-Oriented Systems, Empirical Software Engineering 3 (1) (1998) 65-117.

[20] L. Briand, J. Daly, J. Wust, A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering 25 (1) (1999) 91-121.

[21] L. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, Proceedings of 9th International Conference on Software Engineering, 1997, pp. 412-421.

[22] L. Briand, S. Morasca, V. Basili, Defining and Validating Measures for Object-Based High-Level Design, IEEE Transactions on Software Engineering 25 (5) (1999) 722-743.

[23] L. Briand, J. Wust, J. Daly, V. Porter, Exploring the relationships between design measures and software quality in object-oriented systems, The Journal of Systems and Software 51 (3) (2000) 245-273.

[24] M. Bruntink, A. Deursen, An empirical study into class testability, Journal of Systems and Software 79 (2006) 1219-1232.

[25] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering 20 (6) (1994) 476-493.

[26] F. Dandashi, A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements, Proceedings of ACM Symposium on Applied Computing, 2002, pp. 997-1003.

[27] N. Fenton, Software Measurement: A Necessary Scientific Basis, IEEE Transactions on Software Engineering 20 (3) (1994) 199-206.

[28] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous & Practical Approach, 2nd Edition, PWS Publishing Company, 1997.

[29] M. Fowler, Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999.

[30] B. Geppert, A. Mockus, F. Robler, Refactoring for Changeability: A way to go?, Proceedings of 11th IEEE International Software Metrics Symposium (METRICS'05), 2005.

[31] T. Gyimothy, R. Ferenc, I. Siket, Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction, IEEE Transactions on Software Engineering 31 (10) (2005) 897-910.

[32] R. Harrison, S. Counsell, R. Nithi, An Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Transactions on Software Engineering 24 (6) (1998) 491-496.

[33] S. Henry, D. Kafura, The evaluation of software systems' structure using quantitative software metrics, Software-Practice and Experience 14 (6) (1984) 561-573.

[34] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE Transactions on Software Engineering 7 (5) (1981) 510-518.

[35] IEEE, Standard 1061-1992 for a Software Quality Metrics Methodology, New York: Institute of Electrical and Electronics Engineers, 1992.

[36] IEEE, Std. 610.12 - IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronics Engineers, 1991.

[37] ISO/IEC, 9126 Standard, Information technology - Software product evaluation - Quality characteristics and guidelines for their use, Switzerland: International Organization For Standardization, 1991.

[38] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A Quantitative Evaluation of Maintainability Enhancement by Refactoring, Proceedings of International Conference on Software Maintenance (ICSM'02), 2002, pp. 576-585.

[39] J. Kerievsky, Refactoring to Patterns, Addison Wesley, 2004.

[40] T. Lethbridge, R. Laganière, Object-Oriented Software Engineering: Practical Software Development using UML and Java, 2nd Ed, McGraw-Hill, 2005.

[41] W. Li, S. Henry, Object-Oriented Metrics that Predict Maintainability, Journal of Systems and Software 23 (1993) 111-122.

[42] T. Mens, T. Tourwe, A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126-139.

[43] R. Moser, A. Sillitti, P. Abrahamsson, G. Succi, Does refactoring improve reusability?, Proceedings of 9th International Conference on Software Reuse (ICSR'06), 2006, pp. 287-297.

[44] W. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, PhD Thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[45] R. Pressman, Software Engineering: A Practitioner's Approach, 6th Edition, McGraw Hill, 2005.

[46] W. Salamon, D. Wallace, "Quality Characteristics and Metrics for Reusable Software (preliminary Report)," US DoC for US DaD Ballistic Missile Defense Organization, NISTIR 5459, May 1994.

[47] K. Stroggylos, D. Spinellis, Refactoring - Does it improve software quality?, Proceedings of 5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops), 2007, pp. 10-16.

[48] L. Tahvildari, K. Kontogiannis, A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph, Proceedings of 9th Working Conference on Reverse Engineering (WCRE'02), 2002, pp. 77-86.

[49] L. Tahvildari, K. Kontogiannis, A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations, Proceedings of 7th European Conference On Software Maintenance And Reengineering (CSMR'03), 2003, pp. 183-192.

[50] M. Tang, M. Kao, M. Chen, An Empirical Study on Object Oriented Metrics, Proceedings of 6th International Software Metrics Symposium, 1999, pp. 242-249.

[51] D. Wilking, U. Khan, S. Kowalewski, An Empirical Evaluation of Refactoring, e-Informatica Software Engineering Journal 1 (1) (2007) 27-42.

Vita

Karim O. Elish was born on April 2, 1984 in Al-Khobar, Saudi Arabia and is a citizen of Egypt. He received his Bachelor of Science in Computer Science, with second honors, from King Fahd University of Petroleum and Minerals (KFUPM) in June 2006. In his senior project, he worked in software development project for Saudi ARAMCO. He received three certificates of recognition for this project: the first is from *Saudi ARAMCO* for developing *Project Quality Index System*; the second is from *KFUPM* as distinguished senior project; and the third is from *IEEE - Graduates of the Last Decade (GOLD)* as an outstanding project accomplishment.

Since then, he has worked as a teaching and research assistant in the Information and Computer Science Department at KFUPM while pursuing his MS degree in Computer Science. During the course of his graduate studies, he took advanced software engineering courses such as *Principles of Software Engineering*, *Software Design*, *Software Engineering Experimentation*, and *Software Project Management*. He was awarded the *Outstanding Academic Performance Award* by the College of Computer Sciences and Engineering at KFUPM. He received his Master of Science in Computer Science in June 2008. He is a member of Software Engineering Research Group (SERG) at KFUPM. His research interests include software refactoring, software metrics and measurement, software design, empirical software engineering, and application of machine learning and data mining in software engineering.