EFFECT OF REFACTORING
ON THE PROGRAMMERS' WORKLOAD

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HÜSEYİN CAN DOĞAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2013

Approval of the thesis:

EFFECT OF REFACTORING

ON THE PROGRAMMERS' WORKLOAD

submitted by **HÜSEYİN CAN DOĞAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Ali Hikmet Doğru _____
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Assoc. Prof. Tolga Can
Computer Engineering Department, METU _____

Prof. Dr. Ali Hikmet Doğru
Computer Engineering Department, METU _____

Dr. Cevat Şener
Computer Engineering Department, METU _____

Akif Boynueğri, M.Sc.
Computer Engineer, TÜBİTAK _____

Ali Sağlam, M.Sc.
Computer Engineer, TÜBİTAK _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: HÜSEYİN CAN DOĞAN

Signature :

ABSTRACT

EFFECT OF REFACTORING
ON THE PROGRAMMERS' WORKLOAD

Doğan, Hüseyin Can
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ali Hikmet Doğru

December 2013, 54 pages

In this study, the effects of code refactoring on software programmers' workload were analyzed using a case study.

There were several steps that were followed throughout the analysis. First of all, code smell refactoring techniques related to this project were analyzed based on the studies of Martin Fowler and Joshua Kerievsky. After implementing a few elimination steps, the issues that were good candidates for analyzing the refactoring operation effects on programmers' workload were left. Then, the classes that formed the basis of selected issues were analyzed by looking at their histories and data was collected about related engineering processes. The main aim was to find out the programmers' effort which rose from refactoring and defect solution needs. For quantifying this effort, issue frequencies were calculated for before and after the refactoring operation. After finding the issue frequency, average solution times for issues and issue types was calculated separately for before and after refactoring to find corresponding effort more accurately.

It was revealed at the end of this study that by applying the proposed refactoring techniques

on defined code smells, the effort required by defect solution and other refactoring tasks has decreased.

Keywords: Code smell, Refactoring, Programmer's Workload

ÖZ

KOD İYİLEŞTİRMESİNİN YAZILIMCILARIN İŞ YÜKÜNE OLAN ETKİSİ

Doğan, Hüseyin Can
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ali Hikmet Doğru

Aralık 2013, 54 sayfa

Bu tezde, kodun iyileştirilmesinin yazılım programcılarının iş yükü üzerindeki etkisi, bir durum çalıması kullanılarak incelenmiştir. Bu analiz için iş yükü, kodun yeniden yapılandırılması ve hata çözülmesi için yaratılan iş birimleri araştırılarak ve iş yükü takip sistemi kullanılarak nicelenmiştir.

Analizler sırasında birkaç adım takip edilmiştir. Öncelikle, projedeki sorunlu kod ve kodun iyileştirilme tekniği ilişkisi Martin Fowler ve Joshua Kerievsky'nin çalışmalarına dayanarak araştırılmıştır. Birkaç eleme adımı uygulandıktan sonra, iyileştirme işleminin yazılımcı iş yükü üzerindeki etkilerinin araştırılması için iyi seçimler olan iş birimleri kalmıştır. Sonraki adımda, seçilen iş birimlerinin temelini oluşturan sınıflar, geçmişlerinin izlenmesi ve ilişkili süreç bilgilerinin toplanması için geçmişlerine bakılarak araştırılmştır. Asıl amaç, yeniden iyileştirme ve hata çözümü için yazılımcının eforunun bulmaktır. Bu eforu nicelendirmek için, iş birimi sıklığı kod iyileştirme işleminin öncesi ve sonrası için hesaplanmıştır. İş birimi sıklığı bulunduktan sonra, yazılımcıların eforlarını daha doğru olarak bulmak için iş birimlerinin ortalama çözüm süreleri, refactoring öncesi ve sonrası için ve farklı iş birimi tipleri için bulunmuştur.

Bu çalışmanın sonunda, yazılımcıların hata çözümü ve başka iyileştirme görevlerinden kaynaklanan iş yükünün, belirli kötü kodlar üzerinde önerilen iyileştirme tekniklerinin uygulanmasıyla azaldığı gösterilmiştir.

Anahtar Kelimeler: Kusurlu Kod, Yazılım İyileştirmesi, Yazılımcı İş Yükü

To My Family and My Fiancee,

ACKNOWLEDGMENTS

First of all, I would like to thank to my supervisor Prof. Dr. Ali Doğru for his guidance, suggestions and supports for my master study. His ideas and knowledge helped me a lot to finalize it.

I would like to express my thanks to the jury members, for reviewing and evaluating my thesis.

I would like to thank to TÜBİTAK YTE for supporting my academic works and letting to analyze its project named BÜTÜNLEŞİK.

I would like to express appreciation to BÜTÜNLEŞİK project members for helping me to analyze and interpret their codes and work.

I would like to thank to my fiancee for supporting and helping me throughout the my master study. Her love made me stronger and more determined.

Finally I would like express my special thanks to my family for their trust, patience and love throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Code smell, refactoring and design patterns have always been significant concepts in the software engineering world. Each of them reflects different facets of software and if the programmers want to write reusable, maintainable and readable codes, they should deeply analyze the key parts of them. In other words, these concepts enable to writing better code, to increase the quality of code and to decarease total development effort.

Before all, it has to be stated that the process starts with a code smell. It means that there is a problem with a code that hampers code integrity. After this problem, a refactoring technique comes up. A suitable refactoring technique is selected for each smell for its elimination through identified procedures. Improvement of the software after this operation is in fact another issue. Normally, it is expected that after a refactoring operation, the quality of a modified class increases and maintenance effort and need decrease. Since there is no specific method to show this improvement, interpretation and analysis of the benefits of refactoring is challenging.

"Code smells are the named design anomalies that indicate to a need for refactoring" [1]. By just examining the previous definition, it can easily be concluded that the relationship between code smell and refactoring is quite strong and deep. Provided that duplicated codes and logic, or unreadable code parts, or open for modification-close for extension systems which are encountered, precautions for each are needed to be taken. Every smell has different characteristics and effects; therefore, they have different corresponding solutions. Here, solution means refactoring technique in this context. *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"* [2]. In other words, there is a mapping between these smells and cor-

responding refactoring techniques. By using these techniques, it is easy to remove dangerous and unpleasant effects of the smells and to improve codes' quality attributes.

Martin Fowler declared twenty two code smells [2] at the end of the 2000s. The definitions of those code smells were listed and identified one by one. Moreover, he explained the process in order to solve and fix the smell in a logical order. While adding a new function to the system, fixing bugs, or doing a code review, a programmer can find himself implementing the process which was defined by Martin Fowler. Moreover, Joshua Kerievsky had another point of view for the refactoring issue. He also worked on code smell-refactoring subject; however, he added a third dimension to the subject, which is design patterns. In his book, Refactoring to Patterns, it is stated that: *"Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution"* [3]. Another statement which supports the first one is *"As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context and a certain spatial configuration which allows these forces to resolve themselves"* [3]. He defined twelve code smells in his book. While seven of them exist in the Fowler's list, five of them do not. Nevertheless, he also had a different point of view on the similar code smells. While Fowler analyzed them as simple problems and suggested simple refactoring techniques, Kerievsky dwelled on patterns and suggested pattern-based refactoring techniques.

Both Martin Fowler and Joshua Kerievsky contributed to the field of code smell and refactoring by analyzing the subject from different points of views. That is to say, they both helped the software engineers make their software more maintainable, reusable, and readable. Examining their research and findings, it can be inferred that both scientists clearly defined mapping between code smells and corresponding refactoring techniques. In other words, Fowler defined a code smell set and suggested some simple techniques while Kerievsky defined a smell set by giving pattern-based techniques for each of them.

In addition to the code smell and refactoring subject, the other part of my study is about software quality and especially programmers' workload. Actually, this is closely related with refactoring because refactoring makes the software maintainable, reusable and less error-prone. Furthermore, each refactoring operation eases the later workload of the programmers and the defects and new refactoring requests related with the modified classes are decreased. Especially in the large projects, the developers have to deal with large amount of codes and

classes. Therefore, they have to refactor the software continuously. By doing this, they provide an easier maintenance and growth for the future phases of the project. Although refactoring is closely related with software quality and programmers effort, main aim of this thesis is finding the effects of refactoring on programmers effort and there is no metric or work about refactoring and software quality relation.

In the present study, a real-world project is used in order to collect valuable information about code smells, related refactoring techniques and their effects on later maintenance effort. This project which is called BÜTÜNLEŞİK is a public project which was started in TÜBİTAK in 2009 and it has been under development since then. There are 26 software engineers working in this project. Although most of the project has been completed, some smaller modules are now being implemented continuously. This software is used in every city and every district of Turkey by the employees of Social Assistance and Solidarity Foundations. Technically, there are two main components in the project: client and server. EJB is used at the server side [4]. At the client side, Action Script is used [5]. Naturally, the business logic and the functional parts of the software are at the server side. Actually, it is divided into subparts according to functional modules. There are separate teams and every team is responsible for some of these modules and there are more than 300.000 lines of code existing at the server-side. JIRA is used in the project in order to track and manage the workflow [6].

1.1 Motivation

This study aims to investigate two main topics. First aim of this thesis is to examine the software in the light of *"issues"* in order to extract a mapping between code smells and refactoring techniques. Yet another and more important aim is to analyze the effect of refactoring implementations on the later maintenance effort by using these relations and statistical results.

It is also essential to state that the author has been working on this project for more than two years, and aims to analyze code smell refactoring relation of the project and the effects of refactoring operations on later maintenance effort. During this process, the data was collected and analyzed in the light of the Fowler's and Kerievsky's code smell and refactoring technique relation set. Therefore, at the end of this thesis, we were able to view the improvement of the maintenance effort which was caused by the positive effects of implementing suggested

refactoring techniques to fix code smells.

1.2 Thesis Organization

In the following chapters, the content is organized as follows: In Chapter 2, theoretical information and existing sources that are relevant to this study are explained. In Chapter 3, a summary of Fowler's and Kerievsky's arguments is provided as a foundation for the following chapters. In Chapter 4, the proposed methodology is explained and the steps for the methodology are presented. In Chapter 5, interpretations on the analyzed data are presented in detail. Finally, in the last chapter, a brief conclusion on my research and the introduction of future work are described.

CHAPTER 2

RELATED WORK

2.1 Code Smell and Refactoring Techniques

Code smell and refactoring are two concepts that are studied in relation with each other, in this thesis. Code smell is defined as *"indications that there is trouble [in the code] that can be solved by refactoring"* by Fowler [3]. Refactoring is also defined as *"the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"* [3].

There are many advantages of refactoring for both software itself and programmer. According to Martin Fowler the design quality of the program will decrease without refactoring. He defines refactoring as tidying up the code. He stated that poorly designed code usually takes more code to achieve the same aims. The same code pieces do the same thing in several places. At this point, decreasing the amount of code segments makes a remarkable impact on modification of the code. For instance, bigger code parts render modification harder. Another positive effect of refactoring is that it makes software easier to understand. It is important to consider the future programmer effort and ease it by refactoring the code. Maintaining readable software is easier. Refactoring is used to help programmers understand an unfamiliar code. When they look at unfamiliar code, they have to understand what it does. As the code becomes clearer and easier to understand through refactoring, the programmer is able to see the details of the software design that he could not see before. Another positive effect of refactoring is that it helps the programmers find bugs and defects. The reason behind this is that if they refactor the code continuously, they work deeply to understand the real aim and purpose of the code. Refactoring clarifies the structure of the software and this helps the

programmers clarify certain assumptions they have made. The last advantage of refactoring was described by Fowler who stated that it helps programmers work faster. In other words, it helps them develop the code more quickly. Generally speaking, researchers easily state that refactoring improves the software in terms of many aspects such as design, readability, reducing bugs and so on. All these aspects improve the general quality of software. However, one may ask whether all these additional efforts reduce the speed of development or not. Martin Fowler believes that rapid software development requires good design. Due to the fact that without a good design, the programmers may quickly develop the code, but soon the code structure will become worse and it will decrease the development speed. They may have to start to spent more time to fix the defects and to add new functionalities to the software. These operations take longer because they try to understand the system and figure out the reason behind the duplicated code segments [2].

Joshua Kerievsky also emphasizes implementing refactoring and its benefits. According to him, there are many reasons that motivate the programmers to refactor. First of all, he emphasizes that refactoring makes adding new code or feature easier. When the programmers need to add a new feature to a system, they have two choices. First one is to program the feature quickly without thinking how well it fits into an existing software design. Another option is to modify the existing design so it can easily accomodate the new requirement or feature. According to Kerievsky if the programmers have limited time, it makes more sense to add the feature without modifying the existing design and to refactor later. If they have more time, refactoring the existing design makes sense before adding a new feature to the system. Other advantage of refactoring according to him is that refactoring improves the design of an existing code. The programmers make it easier to work with their code by improving the design and structure continuously. Continuous refactoring in this sense means constant tracking for code smells immediately after noticing them. This also helps them extend and maintain the software easily. Refactoring also enables developers to gain a better understanding of the code since the unclear code parts always cause trouble to the programmers and make their work more difficult. Kerievsky also states that refactoring makes coding less annoying. By applying refactoring, the programmers are able to create human-readable code and to keep it clean so that the development progress will become less annoying [7].

Code smells and refactoring techniques were categorized by Fowler at the end of the 1990s. He defined and listed the code smells and related refactoring techniques one by one. After

that, Kerievsky investigated this subject from a pattern-based viewpoint. He extended the list of code smells and refactoring techniques by analyzing the subject in a pattern-based view. Although the methods that they defined are different, both emphasize the importance of refactoring in terms of many aspects. Having examined these two sources, we can see that code smell and refactoring are considerably relevant with each other.

The relations between code smells have been analyzed by Bartosz Walter and Pawel Martenka. According to them, *"There is an open question if some relations between code smells make also more persistent and reliable patterns, combining anomalies into larger entities. The knowledge about such patterns and their properties could make the search for anomalies more directive and effective"* [8]. Basically they tried to find relationship between code smells. They used Fowler's code smell definitions as the basis of their study. One point that they claimed that one of the code smells may affect another and another point was that some of the code smells generally exist together in the similar context. However, they just focus on the relation between code smells, not the relation between code smells and refactoring techniques.

In another research, again Fowler's bad smell definitions are analyzed. In their study, M.Zhang, N. Baddoo, P.Wernick and T.Hall aimed to improve the precision of Fowler's definitions of bad smells. They said that, first of all Fowler defined a general indication for each bad smell. After that, each indication is divided into several subparts. For each part, Fowler offers several refactoring methods to remove bad smell. They thought that Fowler's bad smell definitions were too informal and they tried to enhance them by using some particular source code patterns. This thesis was just one of the studies on Fowler's bad smell definitions, yet it does not mention the refactoring techniques and their relation to code smells. Neither does it mention the effects of refactoring operations on programmers workload [9].

An empirical study was conducted on subjective evaluation of software evolvability using code smells. For the subjective evaluation of code smells, Fowler's list was utilized again as the source of information for the purposes of the study. However, the aim is not to find a mapping between code smells and refactoring techniques. The main aim is to investigate subjective evaluation of smells according to different developers whose experience, knowledge and opinions are different. Moreover they also compare the subjective evaluations and automatic program analysis' results and they were able to check whether these results were similar or not. Thus, although the main focus of this thesis is Fowler's code smells and the research

is based on a case study, there is no sign or work on their relationship with the refactoring techniques [10].

H.Hamza, S. Counsell, T.Hall and G. Loizou [11] analyzed Fowler's and Kerievsky's code smells and associated refactoring. In their research, they try to calculate the effort for eradicating each of the smells. Therefore, they try to define the complexity and cost of the refactoring by calculating the chains it generates. *"A refactoring chain was first defined by Counsell as the number of refactoring that could be sequentially implemented as part of a higher refactoring."* [11]. In the research the twelve code smells of Kerievsky and twenty-two code smells of Fowler are analyzed. Seven of them are found to be common both in Kerievsky's list and Fowler's list. Actually the possible refactoring techniques are already defined by Kerievsky and Fowler, but in this research [11], the dependency of each technique is defined so that they are able to define the complexity of each refactoring technique. Actually this research includes both Kerievsky's and Fowler's code smells and refactoring techniques. Moreover, it also mentions the relationship between them. However it does not focus on the effects of refactoring on project's and programmers' workload.

2.2 Software Quality, Programmers' Workload and Refactoring Relation

Some of the researches on code smell and refactoring have been mentioned in the previous section. This section will present another important concept that is the quality of code and maintenance effort for the project. Implementing refactoring technique on a code smell improves code quality in terms of many aspects which also improves the workload naturally. In order to identify the positive effects, the software metrics can be analyzed in terms of coupling, cohesion, readability etc. However these concepts are abstract concepts whose quantifications are hard and complicated. Therefore, some methods for quantifying them were developed in the previous research.

For quantifying the effect of code smells on maintenance effort, the following research is investigated. In this study, six professional software developers were hired to perform three maintenance tasks for four Java projects whose owners were different companies. The key point of this study is to measure the time that was spent by each developer in order to maintain the software. They conducted first an empirical study on the code smell effects on software

maintenance effort. Moreover they used a controlled industrial setting. There were many code smells investigated in their research. The maintenance effort for code with smell was quantified and compared with the maintenance effort for code without smell. The main focus was to extract a relation between a specific code smell and its effects on maintenance effort. During their research, they found that files with Feature Envy, God Class, temporary variable used for several purposes, implementation instead of interface and shotgun surgery were associated with more effort than files without such smells. The maintenance effort for files with code smells was bigger than the effort for files without code smells. This means that these type of code smells make the software less maintainable. They also showed that the smell, namely refused bequest was related with a small reduction in effort. Shortly, at the end of this study, they concluded that, maintaining the files with code smells was usually harder than to maintain the files without code smells [12].

Another study was conducted by Konstantinos Stroggylos and Diomidis Spinellis [13]. to find the relationship between software quality and refactoring operation. Their subject was about *"source code version control system logs of popular open source software systems to detect changes marked as refactoring and examine how the software metrics are affected by this process."* Their aim was to find out whether refactoring always improves the code quality or not by analyzing the code by using some software metrics. In their study, they used some metrics to define the quality of software. After that, they tried to show the relation between these metrics and software quality. However they concluded that although it was expected that refactoring improves the software metrics positively, this is not the case for real-time systems. The real work was about examining how the metrics of projects were affected at the end of the refactoring by not at looking the reasons that led to that decision. At the end of this study, they concluded that after refactoring, the classes have become less coherent and their responsibilities were surprisingly increased by looking at the increase rate of the software metrics. In other words according to them, either refactoring operation does not always mean an improvement of code quality or developers have not managed to effectively use refactoring to increase the quality of code [13].

In another research which was conducted by K. Narendar Reddy and A. Ananda Rao, the software quality enhancement was again analyzed by quantitative evaluation. In this research, they used dependency oriented complexity metrics. Three experimental cases were presented in this study and they were analyzed in terms of metrics which represents the improvement

in the quality of code after a refactoring operation. These metrics showed the existence of defects and quality improvement of designs successfully after refactoring operations. Another note about these metrics was that they have acted as design quality indicators of systems that were under small effects before and after refactoring operation and they acted as parameters to define the quality by helping to estimate the maintenance effort. Moreover, software design was studied as a unified representation of *"artifacts graph"* by considering the decoupling of the software. There were many experiments in this study and in all of them, the defined metrics showed the quality improvement for after refactoring. They showed that refactoring eliminates the defect rate and decreases the complexity of structure which meant that the quality of software increased after refactoring. Shortly, at the end of their study, it was shown that the quality of software after refactoring was better than the previous versions [14].

The research that was conducted by Tushar Sharma also focuses on quantifying software design quality in order to find the effects and impacts of refactoring. In this study, it is stated that refactoring increases the total quality of software design. Moreover it is also stated that measuring the quantity of software to estimate the positive effects of refactoring on a software design is a very challenging task. In this research, he introduced a metric namely *"Software Design Quality Index"*. This metric was kind of a metric that enables him to measure the effects of refactoring operation on design of software. His works based on design structures and *"pattern graph"* proposed by Janakiram. While analyzing the refactoring effects, he analyzed desirability of call-patterns in software design and software design quality index. At the end of he defined an automated technique that employed the capability of estimating the quality improvement of software. [15].

CHAPTER 3

FOWLER AND KERIEVSKY'S ARGUMENTS

In this chapter, Fowler's and Kerievsky's points of views are explained. Since their research underlies my research, their code smell list and related refactoring explanations are presented and compared to each other.

First of all, Fowler's arguments are explained. In [2], Fowler first explains the definition of code smell and explains it in detail. He lists twenty-two kinds of code smells in this book. For each smell, he gives a brief description and after that he explains the refactoring techniques to fix these problems. In other words, he offers one or more solutions for each smell separately. Refactoring is also another important topic that is explained by him. Actually, the main part of the book is about refactoring. When he defines the refactoring technique, he uses a standard format that is composed of 'name', 'summary', 'motivation', 'mechanics' and 'examples'. He explains why refactoring should be implemented, a step-by-step description and examples.

Another approach for this issue is Kerievsky's approach. In his book *"Refactoring to Patterns"* he dwelled on code smells and refactoring techniques. However, his perspective is a little bit different. His techniques are not as simple as Fowler's techniques. His solutions involve design patterns and this makes his solutions more deep and difficult. He also describes format of refactoring as the following parts: 'name', 'summary', 'motivation', 'mechanics', 'example' and 'variations'. It is almost similar as Fowler's format. Actually one of Kerievsky's refactoring techniques may consist of more than one of Fowler's techniques. For example; consider the 'Long Method' code smell. Fowler offers six refactoring techniques for this smell and we can see that they are not so complicated techniques. However Kerievsky offers pattern-based solutions for this problem and they may consist of one or more of Fowler's solutions. 'Replace Conditional Logic with Strategy' includes 'Move Method', 'Introduce

Parameter Object' and 'Replace Conditional with Polymorphism' techniques. In other words Fowler's previous researches underlie Kerievsky's refactoring structure.

In the previous parts, the arguments of Fowler and Kerievsky were summarized. The general idea gathered from their perspectives provides us a useful source of information for code smells and related refactoring types. Actually they also provide us a practical mapping which consists of two parts. One part is composed of code smells and the other part is related with refactoring techniques which is either pattern-based or not. In Table 3.1, the code smells and their related refactoring techniques are listed both for Fowler's techniques and Kerievsky's techniques [16]. In this list, there is a total of twenty six code smells. While some of the code smells are related with both approaches, some of them are just related with either Fowler's or Kerievsky's techniques. The common code smells are Duplicated Code, Long Method, Primitive Obsession, Alternative Classes with Different Interfaces, Lazy Class and Large Class. Actually they can be removed by using the related techniques and selection of right techniques is defined by the nature of code and software. While in some cases, just using simple Fowler's refactoring is enough; in other cases the complicated technique of Kerievsky should be used.

Conditional Complexity, Combinatorial Explosion, Oddball Solution, Indecent Exposure and Solution Sprawl are code smells that are only defined by Kerievsky. If there is a code smell in the code that matches any of those, it can be said that the suitable refactoring technique should be the one that creates a pairing with it. For example, if there is *"Combinatorial Explosion"* in the code, it can be said that proper refactoring technique should be *"Replace Implicit Language with Interpreter"*. Moreover this pair definitely should be considered freely without thinking about Fowler's arguments.

The code smell list that is only defined by Fowler (not Kerievsky) is a little bit longer than the Kerievsky's list. There are a total of 15 code smells in this list. They are Long Parameter List, Divergent Change, Shotgun Surgery, Feature Envy, Data Clumps, Parallel Inheritance Hierarchies, Speculative Generality, Incomplete Library Class, Data Class, Comments, Message Chains, Middle Man, Refused Bequest, Temporary Field and Inappropriate Intimacy. These code smells are relatively simple and require uncomplicated techniques.

Table 3.1: Code Smell/Refactoring Types based on Fowler and Kerievsky

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Alternative Classes with Different Interfaces	Rename Method Move Method	Unify Interfaces with Adapter
Combinatorial Explosion	-	Replace Implicit Language with Interpreter
Comments	Rename Method Extract Method Introduce Assertion	-
Conditional Complexity	Introduce Null Object	Introduce Null Object Move Embellishment to Decorator Replace Conditional Logic with Strategy Replace State-Altering Conditionals with State
Data Class	Move Method Encapsulate Field Encapsulate Collection	-

Table 3.1 Continued

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Data Clumps	Extract Class Preserve Whole Object Move Method	-
Divergent Change	Extract Class	Unify Interfaces with Adapter
Duplicated Code	Extract Method Extract Class Form Template Method Introduce Null Object Pull Up Method Pull Up Field Substitute Algorithm	Chain Constructors Extract Composite Form Template Method Introduce Null Object Introduce Polymorphic Creation with Factory Method Replace One/Many Distinctions with Composite
Feature Envy	Extract Method Move Method Move Fields	-
Lazy Class	Collapse Hierarchy Inline Class	Inline Singleton

Table 3.1 Continued

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Inappropriate Intimacy	Move Method Move Field Change Bidirectional Association to Unidirectional Association Extract Class Hide Delegate Replace Inheritance with Delegation	-
Incomplete Library Class	Introduce Foreign Method Introduce Local Extension	-
Indecent Exposure	-	Encapsulate Classes with Factory
Large Class	Extract Class Move Method	Unify Interfaces with Adapter
Long Method	Rename Method Extract Subclass Extract Interface Replace Data Value with Object	Replace Conditional Dispatcher with Command Replace Implicit Language with Interpreter Replace State-Altering Conditionals with State

Table 3.1 Continued

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Long Parameter List	Replace Parameter with Method Introduce Parameter Object Preserve Whole Object	Unify Interfaces with Adapter
Message Chains	Hide Delegate Extract Method Move Method	-
Middle Man	Remove Middle Man Inline Method Replace Delegation with Inheritance	-
Oddball Solution	-	Unify Interfaces with Adapter
Parallel Inheritance Hierarchies	Move Method Move Field	-

Table 3.1 Continued

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Primitive Obsession	Replace Data Value with Object Introduce Parameter Object Extract Class Replace Type Code with Class Replace Type Code with Sate/Strategy Replace Type Code with Subclasses Replace Array with Object	Replace Conditional Logic with Strategy Replace Type Code with Class Replace Type Code with Sate/Strategy Replace Implicit Language with Interpreter Replace Implicit Tree With Composite Replace State-Altering Conditionals with State Replace Type Code with Class
Refused Bequest	Push Down Field Push Down Method Replace Inheritance with Delegation	-
Shotgun Surgery	Move Method Move Field Inline Class	-
Solution Sprawl	-	Move Creation Knowledge to Factory Unify Interfaces with Adapter

Table 3.1 Continued

Code Smells	Fowler's Refactoring Techniques	Kerievsky's Refactoring Techniques
Speculative Generality	Collapse Hierarchy Rename Method Remove Parameter Inline Class	-
Switch Statement	Replace Conditional with Polymorphism Replace Conditional with Polymorphism Replace Type Code with State/Strategy Replace Parameter with Explicit Methods Introduce Null Object	Introduce Null Object
Temporary Field	Extract Class Introduce Null Object	-

CHAPTER 4

IMPLEMENTED METHODOLOGY

In this research, my first aim has been to investigate and analyze the code smell refactoring technique relationship for the subject project. Since the project code that was analyzed is over three hundred thousand lines and there are thousands of issues that have been implemented, I tried to proceed and obtain information in a systematical way. The second aim is to find the effects of refactoring on programmers' workload as maintenance effort. Of course finding the effects is not an easy process. As explained in the Related Work Section, there are many ways to extract this info. Code cyclomatic complexity, dependency of classes, lines of source code, or quantifying the class sizes are some metrices that could be utilized for this problem. In my thesis, the data was analyzed in different perspectives and the *"issue"* concept was used. In other words the issues that are created before and after specific refactoring are main targets of this study.

The quantity of issues and the time intervals show the workload corresponding to the specific class. When a programmer solves an issue and he somehow modifies a specific class, it shows that this class is involved in the solution of that issue and it brings an extra workload to the programmer. Of course all of the issues are not counted in the context of this research. There are many types of issues and each of them corresponds to a different target. Issue types are task, requirement change, topic, defect, decision request, and enhancement. In my research I only considered defect and enhancement types of issues, because they show whether the refactoring improves the workload of programmers. If intensity of these issues decreases after refactoring, this means that this class is involved in less issues for defect solution and enhancement. In other words it shows some success of the refactoring. After refactoring, if the intensity of these issues increases, this means that this modified class actually causes more

trouble after refactoring. In other words the refactoring process did not succeed.

In the next section, the implemented method to extract code smell/refactoring relation will be described step by step. In the latter section, the analysis of the first part result will be shown.

4.1 Extracting Code Smell Refactoring Relation

The issue concept should be explained in detail. First of all our project tracking system that is used in the project is JIRA. By using this tool, we are able to organize and plan our jobs and assignments. Issue is an element in the JIRA which corresponds to a single job item. The types of issues are: task, topic, defect, requirement change, decision request, and enhancement. When there is a smell in the code and if the programmer wants to fix it, an issue whose type is enhancement is created. In fact, most of the issues like this, demonstrate the problematical parts in the software. While some of them correspond to big problems and include many code smells and refactoring techniques, some of them are well-defined and correspond to a single code smell-refactoring relation. Unfortunately, some of them do not contain any valuable and measurable information for my research area. Shortly, in order to achieve my goal, I deeply analyze the issues in JIRA as shown in the Figure 4.1.

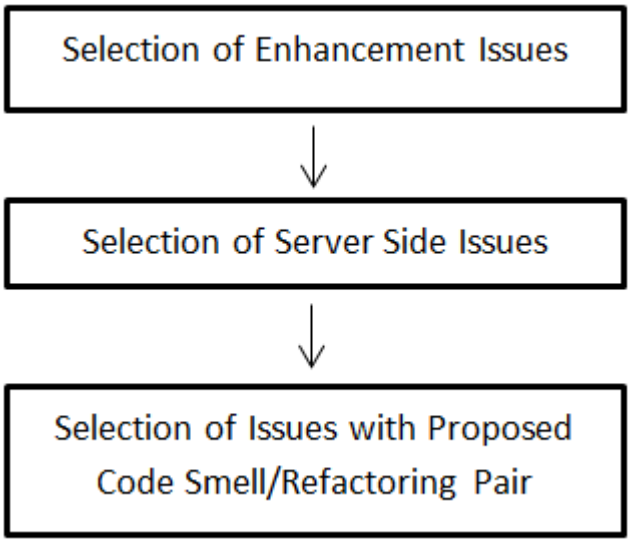

Figure 4.1: Issue Elimination Steps

Firstly, I carefully filter the issues to obtain enhancement issues, because enhancement issues

indicate that there is an improvement in the code either related to my research or not. After first elimination I was able to select 2619 issues out of nearly 34000 issues. However this is just the starting point of my procedure. These 2619 issues of course are valuable for my research, but most of them actually are not a good candidate for analysis for my thesis. The reason is, an important amount of them does not correspond to a traceable and observable refactoring. Actually most of them are created due to changing Software Requirement Specification (SRS), creating new tasks that do not correspond to an actual refactoring, representing the interface or client part modification or changing the database properties and specifications. In other words I have to collect useful and valuable issues within other unrelated issues. The solution that I find for filtering the issues is analyzing them manually by investigating them one by one. It is a little bit long and monotonous process, but I had to do this to obtain important data. After carefully looking at all of them, I extracted 230 issues that are mostly related to a code smell/refactoring technique relation which enabled me to make statistical analyses. After this extraction of issues, the remaining 230 issues gave me a measurable info for my thesis.

In this section, the method that is implemented for an issue is explained. First of all Subversion (SVN) is used in the project and JIRA and SVN are connected with each other [17]. When a commit is done by selecting a specific issue, the modifications can be viewed in the issue window in JIRA. Therefore by using this feature, I can obtain the SVN history of an issue from the starting to the end of the issue process. In this history, all of the code changes that are related to the issue can be viewed. The classes that are added, modified or deleted within the scope of an issue are clearly seen. Therefore my aim is to study all code changes by using history feature of the Eclipse environment [18]. After extracting the history of a class, I found the previous versions of the class. Then I got the point where the class is changed within the scope of a specific issue. The following step of my procedure is just to compare previous class versions for before and after the modification.

Sometimes the issue involves just a few classes and only a simple analysis of their histories is enough to create a result. However sometimes the scope of the issue can be very large and it may involve over hundreds of the code changes. At these points, I try to create subparts of the changes and study them part by part. In other words, I group related code changes in common issue and it enables me to focus on specific parts. While implementing these methods, my reference is always the code smell relation table (Table 1) which represents the code smell

refactoring relations of Fowler and Kerievsky.

When I analyze a code change, first of all I try to understand the code smell that actually shows the starting point of the issue. The defined code smells of Fowler and Kerievsky are already listed and I try to match the code smell to either Fowler's or Kerievsky's smells. Most of the time, it is exactly matched to one of the code smells in the list, but it is a rarely case that I cannot find a match. The reason is either my insufficient analysis for the specific issue or the code problem that does not exactly fit any smell in both lists. Still, I am able to find a code smell and the next part is finding the related refactoring technique or techniques.

Finding related refactoring technique is a little bit harder than finding the code smell. Since there are many alternative techniques for specific smell and analyzing the code for finding the technique is challenging, sometimes this process takes long time. For some issues, I can obtain the technique after studying tens of classes and code changes. Most of the time, I am able to obtain results from issues on my own, because they are small and they include small number of classes. Rarely, I have a difficult time to analyze the issues, because either the relevant and necessary information is not provided with the issue and I have to deal with hundreds of commits or the class hierarchy of the code is a little bit complicated and I cannot fully understand the general structure of the code. At these times, I received support from both code's owners and the experienced developers in the project. Since these experienced developers created the framework and the infrastructure of the code, understanding the code and hierarchy for them is not difficult and challenging.

First of all, the key part for finding related refactoring technique is finding code smell. After finding code smell, I need to specify the group of it. Does it belong to Fowler's list or Kerievsky's list? Since some code smells are common in both lists, finding the correct one required finding the applied technique properly. I carefully research the SVN history of issues and investigate the modified classes. After having full knowledge of the code structure and modifications, the applied techniques become apparent one by one. It is important that whether the technique is in Fowler's or Kerievsky's list or not. Most of the time it is in one of these lists and I finally get a code smell refactoring technique pair. However, it is not always easy, because although some techniques are almost similar in these two lists, finding the difference required deep understanding of refactoring technique. A few times, I cannot find the technique exactly and match any technique in these lists. In such cases, I do not consider the

issue for my research and I just ignore such issues.

4.2 Tracking Subversion History Of Classes

After first part, the list of code smell refactoring relation, the issue number of this refactoring and the main class or classes involved in this improvement are obtained. The remaining part of our process is to analyze the history of the refactored class and the issues that are created before and after refactoring.

The analysis process is expressed in this section. As explained before, the list of code smell refactoring relation, the issue number of this refactor and the main class or classes involved in this improvement are already obtained. Firstly, the classes that are mainly involved in refactoring operation are analyzed. The SVN history of this class is listed and the commits that are related to the refactoring operation are found in the history. In other words the issues that are committed before and after refactoring are obtained. The next part is to look at each issue one by one and find the context and type of it. Actually a timeline is defined for each class involved in refactoring. The center of this timeline is the point which the applied and examined refactoring is implemented. The previous parts of this point are the issues that are created and solved before this specific refactoring is done. The other part includes the issues that are created and solved after the refactoring is done.

In the following, the issues that are involved in this timeline are analyzed one by one. In this part, the important point is the type and the context of the issue. The reason is the main workload of the programmer that came with the refactored class is the issues whose type is either enhancement or defect. In other words these issue types and their quantity represents whether a class causes a lot of work for the programmer or not. Actually these issues reflect whether a refactoring operation is successful or not. Other issue types are totally ignored, because they show either a requirement change, database related issue or a new task related to a new requirement. They are not related with the success rate of refactoring and they did not show the effects of refactoring.

The new data that comes from the previous operation is the main class list that involves refactoring, code smell and implemented refactoring technique, the number of issues (enhancement or defect) that are collected from the SVN history of class for both before and after refactor-

ing. Well, what is the meaning of this data? Actually it means the needed info that shows us whether the refactoring technique reaches its goal or not. From this data, the frequency of issues can be quantified by using the time interval and the number of issues.

$$Issue\ Frequency\ (IF) = \frac{Total\ Number\ of\ Issues(TNI)}{TimeInterval(TI)} \tag{4.1}$$

First of all, the analysis of IF is directly related with the workload of the programmer. Secondly, it reflects the issues whose type is either enhancement or defect. In other words it is related with concepts that enable us to check whether refactoring decreases modification needs of refactored class or not.

The decrease of II result means that the workload of programmers decreases, because it shows that the programmer need for class modification that is caused by defect or enhancement is lower. If the value of II increases, it means that this class causes more work for the programmer. In this study the II results of specific and selected classes are all extracted and the main aim is to show that by implementing the techniques that are already declared by Fowler and Kerievsky, the class and code quality are increased and the workload of programmer is decreased.

By extracting these results, of course there are some challenging parts and points which force to define some specific rules about the procedure. Since the project that is studied is very big and the analysis are done by manual operations, at some points there is need to decide some specific rules and follow them through the study.

As explained before, the main unit of II term is issue count and they can be extracted from subversion history of classes. It can be said that this metric surely enables us to compare the results for separate intervals. Moreover it also shows the improvement of programmers' workload after refactoring. However, although this metric is very important in order to show the effects of refactoring and it is valid, it may not reflect the general workload properly. Since, it is assumed that the workload of an issue as 1 unit and the workload of all the issues are same; defining some time values by using the work log of an issue enables us to compare results more accurately. The problem is that the work log of the issues did not enter fully for under researched issues.

Although the work log of the issues did not enter fully, an estimation based effort is performed

in order to make more correct and accurate evaluations. The important part of this technique is calculating the work log of issues that has a valid work log. Then, an approximate work log is calculated for enhancement and defect based issues separately for first and second time interval. For this process all of the issues that was already founded by tracking subversion history of classes are investigated. Their work logs are controlled one by one. At the end of the investigation, total number of issues that has a valid work log and their total work log time are obtained for each time interval. Since they are also calculated for enhancement and defect type issues separately, at the end of this calculation, four different results are obtained.

$$Average\ work\ log\ (AW) = \frac{Total\ work\ log\ Time(TWT)}{TotalNumberofIssues(TNI)} \tag{4.2}$$

In the above equation, AW means the average time that spent for an issue. At the end of this process, four different AW values are calculated for representing two different time intervals and two different issue types. TWT is obtained by calculating the sum of all selected issues that has a valid work log. The last metric TNI is total number of selected issues that has a valid work log.

4.3 Exceptional Cases

Infrastructure Code Effect

In the project that is analyzed, there are some modules which assigned to specific teams and their members. Each team is responsible for specific modules and their members are solving the issues of about them. However at the center of the project, there is a big and important module whose name is infrastructure. Actually, there is a team which is responsible for the infrastructure of the software and other teams' code parts depended on this infrastructure code.

The relation between the infstructure code and our study is about the code changes of infrastructure code. As explained before, for this study the history of classes are analyzed and the number of issues whose type is enhancement and defect are calculated. However for this calculation, the infrastructure code are omitted. The reason is, even if the small changes are done in infrastructure code, then this affects all of the modules and this modification can be

viewed in the history of these classes. Actually these type of code changes did not reflect the code quality of other module's classes which is the main part of this study. The main aim is to investigate the workload of the programmer for before and after the specific refactoring to find the effects of it, but the changes that came from infrastruture are irrelevant to refactoring and its effects.

Client side refactoring issues

The core part of this research is to track the history of some selected classes and client side refactoring issues is another topic that affected the procedure and implemented method. While tracking the history of these classes, issues whose type are enhancement and defect are calculated, but client side refactoring issues are also an exception case like infrastructure issues. Since the main research area for study is server side EJB codes and classes, client side refactoring issues did not reflect the effect of actual refactoring operation. Shortly, these types of issues were not included in this study.

Requirement Change and Task Type Issues

Another exceptional issue types are requirement change and task. Actually the main aim is to find the workload of the programmers that is caused by refactoring and defect. Therefore, there may be issues which are created for new additional features of the project or some new tasks that are assigned to the programmers. These types of issues are created both before and after the analyzed refactoring issue and they did not contain any valuable information for this study.

Multiple Commits for Analyzed Issue

While analyzing the issue and related history, there are some cases for which there are multiple commits for both the analyzed issue and other issues. For these cases, some rules are defined in order to calculate the results more accurately.

If there are multiple commits for an investigated issue, the intermediary commits are ignored. Since the analyzed issue is partially finished, it is not correct to calculate intermediary issues. Shortly for these cases, the end date of the first interval is accepted as the start date of inspected issue, and the start date of the second interval is accepted as the end date of it.

If there are multiple commits for a single issue in the history of class and if these commits are

not the analyzed issue, it is incorporated as just one single issue for the calculations in this thesis.

Multiple History Analysis For Single Class

In this study, the history of selected classes is analyzed according to the refactoring type and date for programmer's effort. In large projects, it is very natural to apply multiple refactoring techniques that are suggested by the previous studies. Since the aim is to track the history of these issues, sometimes the history of the some classes are investigated more than once for different commits. Actually the main point of this study is to show the influence of refactoring on programmers workload. Therefore for these cases, the history of one single class is investigated for each refactoring and each result is included in this study.

Package Refactoring

Package refactoring is actually a simple refactoring type to implement, but it affects many classes in the project and appears in the history of them. In this project, there are some issues whose main aim is just to rename the packages in the direction of new features and requirements. While analyzing the history of selected classes, these types of issues are ignored and are not calculated. In other words these types of issues did not give any valuable information for the classes maintenance and programmers' workload whose source is the refactoring operation under investigation.

Renaming Refactoring

There are some issues in the project whose concern is just to rename some methods or classes because of new requirements and domain terms. Of course these types of issues are seen in the history of the selected classes which are modified for the investigated issues, but there is no direct relation between these two issue types. The commits that are made for the renaming aim are totally irrelevant to the analyzed refactoring issue and its effects on the classes and programmers' workload.

Commits without Issue Number

In the history of some classes, there are some commits which did not relate to an issue in JIRA. Although cases like this are not too much, a specific rule is defined for them: the content and the explanation of the commit are read. If the explanation shows that a refactoring

is implemented or a defect is solved, then this issue is added to the context of this study and calculated as a single issue. Of course, another criterion that is followed is the content of the commit. Only server-side modifications are included to the study.

Finding Candidate Classes

Before tracking the history of classes, they should be carefully and clearly defined by analyzing the context of the selected issue. Sometimes there are many modifications which are related to it, however analyzing the history of all of them is not the right idea for finding the candidate classes.

First of all refactoring is conducted in order to solve some specific problem of the code. The context is clear and the problematic classes are already defined by relating them to the code smell. In this study, the starting operation is to find the issues which are arisen from a refactoring need to handle some code smell. The classes that constituted the core of the refactoring are already appearant. Therefore, after finding the right issues, they are filtered by conducting the previous criteria.

Issues Starting Date

Sometimes during this study, some exceptional cases were viewed. While analyzing the history of a class, the commits are ordered by the number of SVN commit. Most of the time, it shows parallelism with the starting date of the related issue. However in some exceptional cases, although starting date of the issue is before the analyzed refactoring, it is solved after it. It means that these types of issues should be analyzed by considering their creation dates, because the need of modification or error-fix is before the creation date for the analyzed issue. Therefore, it should be viewed for its own time interval.

Misleading Enhancement Issues

In this study, it is stated that enhancement issues that signal for refactoring and defect issues are very critical and important for our calculations. However, adding every enhancement issue to our data pool is a little bit risky, because it has been observed that many issues whose type is *"enhancement"* actually referred to a new task or a new requirement set. Unfortunately the type of these issues were entered wrongly, but of course they are eliminated and are not included in this research.

Issues Not Related With SVN

In the project, the programmers had normally committed their code by relating it to the issue under research. In the analysis part, sometimes it can be seen that some issues that are not related with the SVN. In other words, programmers committed their code for an issue, but they did not relate it with the issue itself. For these situations, the class history does not contain a commit that corresponds to an analyzed issue whose commit time was the center of the time interval. The solution that is found is to examine the issue through a fixed date and accepting it as middle of the time interval.

CHAPTER 5

RESULTS AND EVALUATIONS

In this chapter the evaluation metrics and evaluation of results will be shown.

5.1 Evaluation Metrics

5.1.1 Issue Frequency

Issue Frequency is a metric that shows the frequency of issues whose type is enhancement or defect in a specific time interval. Normally there are two time intervals for a specific class that is analyzed under the specific refactoring issue. First time interval corresponds to the time that reflects before refactoring time. Second time interval corresponds to the time that reflects after refactoring time. The frequency is calculated for both time intervals for a specific class and at the end of the study, all of the data were collected in order to reach a reasonable result.

Total Time Interval represents a time interval for calculating the frequency before and after the refactoring time. As explained before, each class has two time intervals and issue frequencies differ according to them. First interval starts with the commit date of the first issue that is related to a class and it ends with the commit date of the refactoring issue under investigation. Actually the starting point of the second interval is same as the end date of the first interval. Finally the end date of the second interval is accepted as the current date for the research time.

Total Number Of Specific Issues shows the total number of issues which are created during the mentioned time intervals. These issues actually represent whether there is a commit related with the class or not. However this commit actually means a special type of commit which corresponds to either another refactoring issue or a defect related issue. These issue types

show the workload of programmer that should be improved. Expected result is with each refactoring implemented on a class, the frequency of these types of issue should decrease.

Table 5.1: Results

Results	Interval 1 (Before Ref.)	Interval 2 (After Ref.)
Total # of Days	33.834	22.196
Total # of Issues	475	163
Total # of Enhancement Issues	355 (%74.7)	138 (%84.6)
Total # of Defect Issues	120 (%25.3)	25 (%15.4)
Issue (Total) Frequency (issue/day)	0.0140	0.0073
Issue (Enhancement) Frequency (issue/day)	0.0105	0.0062
Issue (Defect) Frequency (issue/day)	0.0035	0.0011

5.1.2 Average work log

Average work log and the following metrics are calculated for improving the results that were already described in terms of *"IF"*. As explained before, IF shows the frequency of issues and it can be used to compare the workload between time intervals. However by using this metric, it is assumed that the workload for issues are all equal for the programmers. In order to evaluate results more accurately, average work log is calculated. Actually, it shows the work log for completing a single issue.

Total work log Time is obtained by calculating the sum of all issues whose work logs were correctly entered. While calculating this metric, the unit is selected as minute and all of the operations are done based on this unit.

Total Number of Issues This metric does not represent all the issues that were selected in the tracking operation of class histories. It actually represents the issues that have a valid work log that shows the time correctly that was spent by programmers. In order to extract this value, all of the issues are analyzed in JIRA and more than half of them are eliminated for their invalid and missing entries.

$$Average\ work\ log\ per\ issue\ (AW) = \frac{Total\ work\ log\ Time\ (TWT)}{Total\ Number\ of\ Issues\ (TNI)} \tag{5.1}$$

Table 5.2: Analysis Results for Enhancement type issues before refactoring

Results	Interval 1 (Enhancement)
Total work log Time	2692 h 24 min
Total # of Issues	187
Average work log	864 min (14 h 24 min)

Table 5.3: Analysis results for defect type Issues before refactoring

Results	Interval 1 (Defect)
Total work log Time	615 h 57 min
Total # of Issues	42
Average work log	880 min (14 h 40 min)

5.1.3 Estimated Programmer Effort

The results are given for both issue frequency and average work log. The results of issue frequency produce convincing information to indicate the effects of refactoring on programmers' workload. However for improving the validity and reliability of the results, average work log results are also used. Issue frequency shows the commit frequency for a single day. Since each commit is related with an issue, the relationship is actually based on issues. Moreover by using average workload, it can be identified as the time value for the completing of a single issue. Therefore, the frequency values are also made to relate with time duration values instead of remaining as only issue counts.

$$Estimated\ Programmer\ Effort\ (EPE) = IssueFrequency(IF) \times Averageworklog(AW) \quad (5.2)$$

Table 5.4: Analysis results for enhancement type Issues after refactoring

Results	Interval 2 (Enhancement)
Total work log Time	1175 h 32 min
Total # of Issues	113
Average work log	624 min (10 h 24 min)

Table 5.5: Analysis results for defect type Issues after refactoring

Results	Interval 2 (Defect)
Total work log Time	108 h 56 min
Total # of Issues	20
Average work log	327 min (5h 27min)

Table 5.6: Average work log results

Results	Interval 1	Interval 2
Total work log Time	3308 h 21 min	1283 h 28 min
Total # of Issues	229	133
Average work log	867 min (14 h 27 min)	579 min (9 h 39 min)

$$Total\ EPE(Interval\ I) = EnhancementEPE(Interval\ I) + DefectEPE(Interval\ I) \quad (5.3)$$

EPE is the actual metric that shows us the quantified programmers' workload in terms of observable results. As shown before, IF can also be used to show it, however results of EPE are more reliable than results of IF. First of all, EPE includes two different metrics. The first metric is issue frequency which shows the frequency of issues. In other words, IF means the issue rate that corresponds to a single workday. However, explaining the workload by just using issue concept may misguide us, because it assumes that the workload corresponding to all kinds of issues is same. Therefore, the second metric, which is AW, is added for our calculations. AW corresponds to a average time spent that is used for resolving a single issue.

While calculating AW and EPE, the result set is divided into four. They are Interval 1 (Enhancement), Interval 1 (Defect), Interval 2 (Enhancement) and Interval 2 (Defect). There are two reasons that explain the logic of this classification. First of all, the nature of time intervals is different. While interval 1 corresponds to a time that reflects left side of the refactoring point, interval 2 shows the right side of the refactoring. Moreover the comparison between the results of interval 1 and interval 2 underlies the framework of this study. Secondly, the behavior of enhancement and defect issues are also different. The reaction of their results after refactoring is also different which can be seen in the Figures 5.1 and 5.2.

The unit of EPE is simple which is minute/day. It directly refers to a time value which is

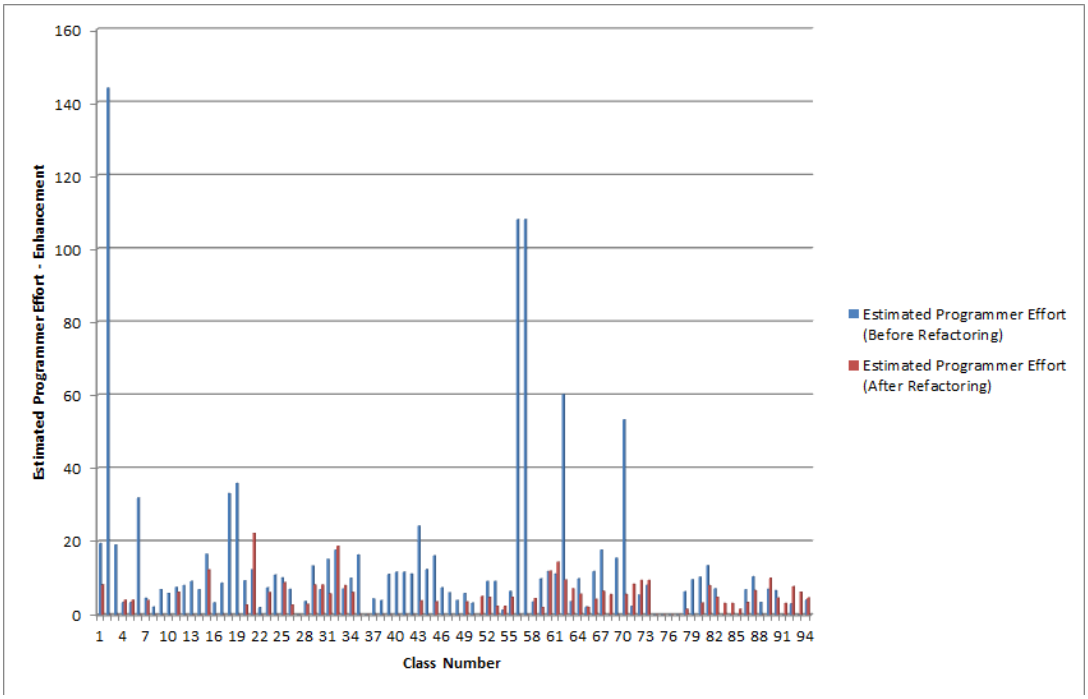

Figure 5.1: Programmer Effort values related with enhancement type issues

spent by a programmer in one day. The unit of EPE actually depends on the unit of IF and AW. The unit of IF is issue/day which means the number of commits (a commit relates with an issue) that is made for refactored classes in a day. The unit of AW is minute/issue which stands for the time spent for an issue. In the following tables, it can be seen that IF and AW values are calculated separately for interval 1(Enhancement), Interval 1(Defect), Interval 2(Enhancement) and Interval 2(Defect). The reason of this calculation is that the commit frequency and resolving time of issues are different for these metrics. Therefore, instead of calculating the resolving time of defect and enhancement issues together, the calculations are made separately for each of them and it makes the result more reliable and accurate.

Table 5.7: EPE Results

Results	Interval 1 (Enhancement)	Interval 1 (Defect)	Interval 2 (Enhancement)	Interval 2 (Defect)
IF (issue/day)	0.0105	0.0035	0.0062	0.0011
AW (min/issue)	864 min (14 h 24 min)	880 min (14 h 40 min)	624 min (10 h 24 min)	327 min (5h 27min)
EPE (min/day)	8.98	3.08	3.87	0.36

Table 5.8: Total EPE Results

Results	Interval 1	Interval 2
Total EPE (min/day)	12.06	4.23

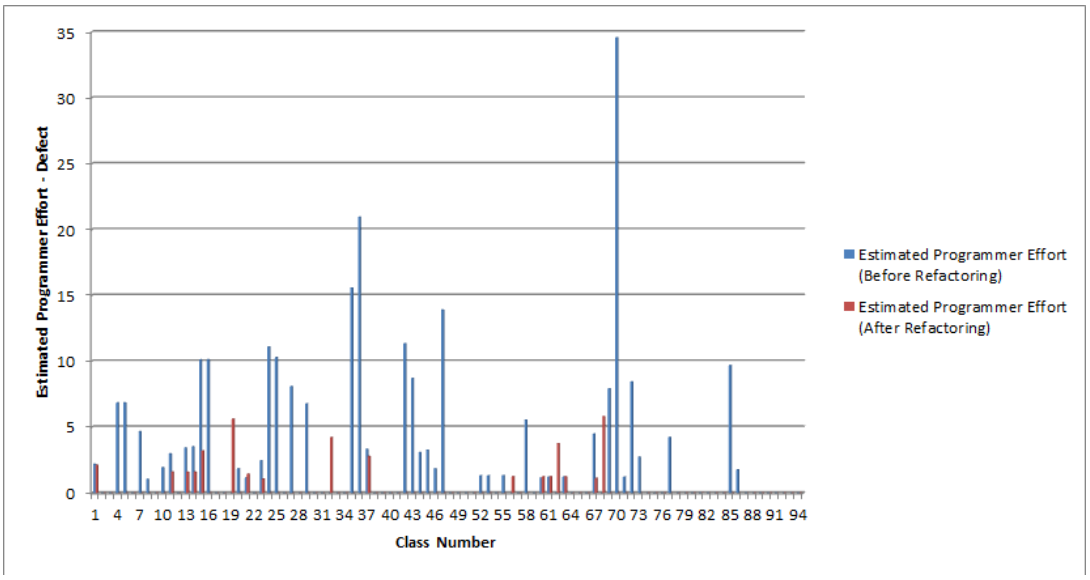

Figure 5.2: Programmer Effort values related with defect type issues

5.1.4 Statistical Analysis of Estimated Programmer Effort

In the previous sections, the results of EPE were shown. However, there was no statistical analysis in the previous sections. In this sections, the results of EPE are analyzed and t-test is implemented for these results. Values for enhancement and defect data sets are processed seperately. These sets are compared in order to show whether there is significant difference between their *"before enhancement"* effort and *"after enhancement"* effort or not.

First of all there are two different types of data. One of them is for enhancement results and other is for defect results. Each data contains two data sets which represents the results for before and after refactoring operation. Before starting the T-test, our null hypothesis should be defined. The hypothesis is that there is no significant difference between the results for before and after refactoring. Moreover the significance level is selected as 0.05 as a common valur for most such studies. T-test is also implemented for enhancement and defect data sets seperately. In table 5.9 the P-values are shown. Since P-values are less than 0.05, the null

hypothesis for both data sets were rejected. In other words, there is significance difference between the results of enhancement and defect issues for before and after refactoring.

Table 5.9: Statistical Analysis of EPE by the T-test method

Results	Enhancement Data Set	Defect Data Set
P-value	0,00015086078668	0,000000000177482

Another statistical analysis for EPE results was implemented to find mean, standard deviation and sizes of data sets. In table 5.9, the mean, standard deviation and size values for four different data sets are shown. One important point is that the standard deviation of enhancement data set for before refactoring is high. In the next section this result is updated after excluding the extreme values.

Table 5.10: Statistical data on EPE

Results	Enhancement Data Set (Before)	Enhancement Data Set (After)	Defect Data Set (Before)	Defect Data Set (After)
Mean	12.87268085	3.704170213	2.826297872	0.435536170
Standard Deviation	22.13974835	4.338636094	5.279508823	1.138190877
Data Size	94	94	94	94

5.1.5 Improvement Ratio

In the previous sections, EPE was described as the main metric of this study but in order to show the real effort improvement, a ratio is also included. This metric shows the improvement ratio for both defect and enhancement types of issues.

$$Improvement\ Ratio\ (IR) = \frac{EPE\ (After) - EPE\ (Before)}{EPE\ (Before)} \tag{5.4}$$

For some of the EPE results, the improvement ratio value was also calculated. However, the improvement ratio results could not be calculated for the results whose EPE (before) values were zero. 13 improvement ratio results for enhancement and 51 improvement ratio results for defect could not be calculated. For this reason, data for such cases do not exist in Table

5.11, modifying the statistical results. However, after excluding this small number of samples, the results are not affected significantly.

Table 5.11: Improvement Ratio analysis

Results	Effect Improvement (Enhancement)	Effect Improvement (Defect)
Mean	-0.468212432	-0.79957333
Standard Deviation	0.67257588	0.370280259
Data Size	81	43

Normally, mean values for improvement ratios are negative for the improvement of programmers' effort; improvement corresponds to reduction. This result shows the importance of refactoring in terms of programmers' effort.

5.1.6 Statistical Analysis After Excluding Outliers

In the previous sections the statistical results were shown for both EPE and Improvement Ratio results. Since, some of the EPE values did not parallel with the general behavior of the data set, the data samples with the maximum and minimum 3 values were eliminated for each of these four data sets in order to show the results more exact and accurate. Especially, the standard deviation value for the enhancement data set for before refactoring were too high as explained before. However, after eliminating the extreme values, mean and standard deviation results resemble a more reliable data set. This change also affected the improvement ratio results and the new results can be seen in tables 5.12 and 5.13.

Table 5.12: EPE data after filtering

Results	Enhancement Data Set (Before)	Enhancement Data Set (After)	Defect Data Set (Before)	Defect Data Set (After)
Mean	9.660109091	3.326345455	2.211000000	0.287239773
Standard Deviation	10.11579478	3.407025523	3.450089521	0.740787247
Data Size	88	88	88	88

Table 5.13: Improvement Ratio analysis after filtering

Results	Effect Improvement (Enhancement)	Effect Improvement (Defect)
Mean	-0,433436859	-0,807468012
Standard Deviation	0,840188618	0,352125565
Data Size	78	40

5.2 Evaluation of Results

5.2.1 Total Number of Days

In the previous sections, time interval concept was explained. For interval 1, total #number of days meant sum of the first intervals for every class that was under research. The logic was also applied for interval 2, same. The results showed that total number of days for interval 1 and interval 2 were 33.834 and 22.196 respectively.

It could be clearly seen that interval 1 result was bigger than the interval 2 result. The main reason of this difference was the selected refactoring and their implementation dates. Actually the issues that were selected before them were mostly new issues, because their validity and data were more reliable than the older issues. Since their creation dates were not too old, it was natural that interval 1 was bigger than interval 2.

5.2.2 Total Number of Enhancement and Defect Issues

These metrics showed the number of enhancement and defect issues for interval 1 and interval 2. As explained before, these issue types were either enhancement or defect and this metric reflected their total numbers and their percentages. It actually helped us to make comparison between enhancement and defect issue types.

First of all the percentages of enhancement issues were %74.7 and %83.8 which helped us to make two different comments. First of all, the percentage of enhancement issues were clearly bigger than defect issues which shows that the programmers mainly worked enhancement type issues rather than defect issues when considering these two types. Secondly, after an important refactoring, the decrease in defect issues were bigger than the decrease in enhancement issues. This is the main reason of this difference for interval 1 and interval 2.

5.2.3 Total Number of Issues

It could be seen that total #number of issues for interval 1 and interval 2 was 475 and 163 respectively. Since total #number of days for interval 1 was bigger than interval 2, it could be expected that total #number of issues for interval 1 was bigger than interval 2 at the same ratio. However there was another criterion which formed the structure of this study. This criterion was that while interval 1 reflected before refactoring time, interval 2 reflected after refactoring. The expected thing is that the frequency should be smaller for interval 2 which will be explained in the next sections.

5.2.4 Average Workload

After combining this metric with IF, quantifying the programmers' effort became more valid and meaningful. AW was calculated for four different metrics where the interval and issue types were considered.

The results of AW values showed that they really differed for interval 1 and interval 2. Actually, this was the expected case. The reason is that interval 2 corresponds to after refactoring. In other words, in interval 2, it is expected that solving defect or enhancement issues should be more easy and less time consuming.

AW results for Enhancement kind of issues in interval 1 and interval 2 were 14h 24min and 10h 40 min, respectively. It means that, in the refactored class history, the effort spent for resolving enhancement issues that were related with the commits in SVN were improved. In other words, the programmers' workload was decreased in terms of resolving time.

AW results for defects were in the same direction with the AW results of enhancements. AW result for Defect type issues in interval 1 was 14h 40min and AW result for interval 2 was 5h 27min. The improvement related to defect issues was better than the improvement related with enhancement issues. In other words it was shown that applying suggested refactoring techniques decreased the workload that are related with the refactored class after refactoring. However, this improvement was bigger for defect type issues.

The average workload values for interval 1 and interval 2 were also calculated for just giving information. However, they were not used for calculation of estimated programmer efforts.

The reason is, AW values were already calculated for four different cases separately based on time period and issue types and the issue frequency for each case is different.

5.2.5 Total Issue Frequency

Maybe one of the important result metric for this study was total IF, because it actually showes the importance of implementing suggested refactoring techniques which improves the software structure and helps programmers to ease their workload. Bigger IF values means more workload for the programmers. Smaller IF values means the workload was less.

The most important comparison for this metric is of course between the interval 1 and interval 2. It created the motivation of this study and produced important results to us. IF value for interval 1 was 0,0140 which meant the commit frequency that was related with enhancement or defect issue for a single day. Actually at this part, the type of issue was not so critical, because both of these types point a workload for the programmer. The result for interval 2 was too important and informed us about the effects of refactoring implementations on programmers' workload. The result was 0,0073 for interval 2 which means the commit frequency related to enhancement or defect issues was 0,0073 for a single day. Well, what is the meaning of these values? First of all, it could be easily seen that IF value significantly decreased for interval 2. This means that the programmer needed to handle defect or enhancement issues less likely in interval 2, which also implies that the software quality was increased. In other words, the need for enhancement or defect solution was decreased. All of these results actually can be used to conclude that implementing suggested refactoring techniques made the software more maintainable and decreased the enhancement and error solution need of the programmer significantly.

5.2.6 Enhancement Issue Frequency

One of the important metric was *"total issue frequency"*; however this and the next metric gave also an important information about the refactoring effect on enhancement and defect solving needs separately. First of all, enhancement issue frequency was 0,0105 for the interval 1, and 0,0062 for the interval 2. In other words for interval 1, the programmer's commit frequency for a single day was 0,0105. In interval 2, this value was 0,0062 which shows a

decay in the software workload for enhancement type issues. The decrease in this metric was clearly seen like total issue frequency.

5.2.7 Defect Issue Frequency

Another metric was defect issue frequency which shows the frequency of commits related with defect type issues for a specific time interval. The values were actually more impressive than enhancement frequency values. For interval 1, the frequency value was 0,0035 which shows the commit frequency of class history for a day. This value may be seen lower when compared to the previous values. However the value for interval 2 was very interesting. The value was 0,0011 which meant that the commit frequency with defect issues for under researched classes was 0,0011. This result was very impressive. The workload that was caused by defects and errors were significantly lower after refactoring implementation. This result also shows the importance of applying suggested refactoring techniques for the future of software pertaining to both programmers' workload and software quality.

5.2.8 Estimated Programmer Effort

Although IF and AW showed important results, it was more meaningful to combine these two metrics and to extract more valid results. EPE was a metric that showed programmer effort with the help of the IF and AW metrics. The meaning of programmer effort was the workload of the programmer that was caused by enhancement or defect issues. The main idea was to show that this workload decreased after implementing refactoring techniques.

Since IF and AW results were calculated for four different cases , at the end of this research four different EPE results were obtained. These results differed in terms of refactoring interval (before or after the refactoring time) and resolved issue type (enhancement or defect type issue). The results of EPE were 8.98 , 3.08 , 3.87 , 0.36 which corresponded Interval 1(Enhancement), Interval 1(Defect), Interval 2(Enhancement) and Interval 2(Defect).

The meaning of the previous EPE results will be explained in this section. First of all, the meaning of EPE was total spent time. (in minutes per day) by programmers for resolving issues whose related commits appeared in the related classes. For example; the EPE result for Interval 1 (for Enhancement type of issues) was 8.98. In other words, the programmer spent

8.98 min/day for solving enhancement issues before refactoring where this type of issues were related with commits for the related classes. The EPE result for Interval 1 (for Defect type of issues) was 3.08 which meant the programmer spent 3.08 minutes per day for resolving defect issues before refactoring. Moreover these issues were related to commits which were observed for the related refactored classes. We had 2 more results representing the time interval 2. Interval 2 (for Enhancement type of issues) EPE result was 3.87 and Interval 2 (for Defect type of issues) result was 0.36.

When we compare the results, it can be seen that the result for enhancement was decreased from 8.98 to 3.87. This is an important assessment, because it shows that total resolving time for enhancement issues which were related with commits for under researched classes were significantly decreased. In interval 1, the programmer needed to spend 8.98 minutes per day for solving these type of issues, but the new value was 3.87 after refactoring. This also showed that the workload of the programmer was decreased and the main reason for this decrease was the success and positive effects of applied refactoring techniques.

One more comparison could be investigated that is the comparison of results for Interval 1 (Defect) and Interval 2 (Defect). Interval 1 (Defect) EPE result was 3.08 and Interval 2 (Defect) result was 0.36. In other words, in interval 1, a programmer spent 3.08 minutes per day for resolving defect type issues, but only 0.36 minutes after refactoring. This showed that the programmers workload dedicated to defect issues for the refactored class decreased significantly. This comparison also shows the importance of refactoring for its positive effects on programmers workload.

In this section, the positive effects of refactoring were quantified. The EPE metrics was used. The relation between programmers' effort on defect or enhancement types of issues and refactoring was shown. EPE directly showed the effort that was spent by programmers on related classes. Its unit was minutes per day which gave us a clear and understandable result.

5.2.9 Analysis of Normal Classes

In the previous sections, it was shown that issue frequency and estimated programmer effort results decreased after refactoring. In the analyzed software, some of the classes whose histores did not have a commit which was related with the proposed refactoring issue were later

analyzed to obtain control data. Their issue frequencies were calculated by considering the rate of time intervals for before and after some time that corresponded to the refactoring time for the experimentation group. The total number of period lengths for previous calculations were 33.834 and 22.196 for before and after refactoring respectively. Therefore, in order to calculate the issue frequencies for classes that were nor refactored, the values for the previous time intervals were used. For the total period of a class in the control study, part of this period was taken as *"Period 1"* based on the same ratio between the periods 1 and 2, calculated from the average values of the refactored classes under study. In other words, since total time interval for this calculation was 12.180, the first interval equals to 7308 days and the second interval equals to 4872.

First of all, 20 classes that were not refactored by proposed refactoring techniques selected randomly. Then, their histories were analyzed in order to find the commits which were related with defect and enhancement issue types seperately. Moreover, the rules for exception cases in 4.3 were also applied for this calculation again.

$$First\ Interval = \frac{12.180 * 33.834}{33.834 + 22.196} \tag{5.5}$$

$$First\ Interval = 7308 \tag{5.6}$$

$$Second\ Interval = \frac{12.180 * 22.196}{33.834 + 22.196} \tag{5.7}$$

$$Second\ Interval = 4872 \tag{5.8}$$

In the previous table, only issue frequencies were shown. It can be said that the issue frequencies were almost equal for interval 1 and interval 2. Normally, in order to find EPE, average worklog values should also be used, however the main aim of this section is to show the distribution of issue frequencies for normal classes. By looking at the average worklog results for refactored classes, it can be said that the results for after enhancement are smaller than the results for before enhancement. Although data could not be collected for the mentioned purpose, the same trend is not expected in the control classes group.

Table 5.14: Values for the control group of data

Results	Interval 1	Interval 2
Total # of Days	7308	4872
Total # of Issues	43	30
Total # of Enhancement Issues	34 (%74.7)	23 (%84.6)
Total # of Defect Issues	9 (%25.3)	7 (%15.4)
Issue (Total) Frequency (issue/day)	0.0058	0.0061
Issue (Enhancement) Frequency (issue/day)	0.0046	0.0047
Issue (Defect) Frequency (issue/day)	0.0012	0.0014

5.2.10 Comparison with the Control Group

Based on the IF values, a comparison with the control group can be made. The final result of interest is the improvement therefore a comparison between the improvements of the refactored and the control group of classes will be made. If we can define the improvement in IF for the refactored group as:

$$I_{IFr} = \frac{(IF_{r2} - IF_{r1})}{IF_{\mathrm{r1}}} \tag{5.9}$$

Where IF_{r1} is the IF for the refactored classes in Period 1 and IF_{r2} is the IF for the refactored classes in Period 2.

The improvement in IF for the control group as:

$$I_{IFc} = \frac{(IF_{c2} - IF_{c1})}{IF_{\mathrm{c1}}} \tag{5.10}$$

Where IF_{c1} is the IF for the refactored classes in Period 1 and IF_{c2} is the IF for the refactored classes in Period 2.

The values for I_{IFr} and I_{IFc} are -0.4785 and 0.0003 respectively. As can be seen, the improvement is significantly better in the refactored group when compared to the control group.

5.2.11 Comparison of Studies

There are few studies in the literature that are related with this thesis. In the study of Sjoberg et. al, [12] the metric that is used to quantify the effect of refactoring is maintenance effort that is spent by programmers. The main focus of that study is to compare the maintenance efforts for the classes with and without code smell. This metric actually looks similar with the EPE. It represents the programmers' effort on defect or enhancement type works, but in [12] the maintenance effort is calculated directly. Secondly, EPE is calculated by tracking the quantitive values of a real world project. Moreover nearly 25 software programmers' code was analyzed, but in [12] only six developers are hired to perform coding operations. The main aim of the mentioned research [12] is to show that maintaining the code with smell is harder than the code without smell. However in this thesis, EPE shows that after refactoring, the workload of the programmer that is caused by defect or enhancement related refactoring, decreases.

In another study [13] again the effects of refactoring operation are quantified. In the study, the relation between software quality and refactoring is defined and analyzed. The strategy that was defined is different. It depends on to track source code version control system logs in order to find how some software metrics are affected after refactoring operation. Actually the aim is to find whether refactoring operation always improves the quality of software or not. Metrics were also used for this study and their relation with software quality was defined. At the end of the study, it is shown that classes have become less coherent and the responsibilities of them were increased. In this thesis, positive effects of refactoring is shown by relating it with the workload of the programmer, but in [13] the main aim is to define the relation between software quality and refactoring and at the end of it, negative aspects of refactoring were shown.

In the study by K. Narendar Reddy and A. Ananda Rao, [14] enhancement of software quality was researched. Moreover, quantitative evaluation was used. The metric used to quantify the improvement of the software was dependency oriented complexity metrics. These metrics are used to identify the existence of errors and improvement of design quality after refactoring. These metrics are also used to express the improvement of the quality which helps to estimate the work for the maintenance effort. Their study is similar with this study for showing the decrease of the defect rate after refactoring. However they based their study on analyzing

three experimental cases and used their own metrics. They showed that the workload that was caused by defect is increased after refactoring by analyzing class histories and tracking work logs. Another criterion of their study is to investigate the improvement of software quality, but in my research, workload of programmers is analyzed.

Table 5.15: Comparison Table

Study	Main Aim	Context Variables	Dependent Variables	Analyzed Systems	Is Programmers' Effort Analyzed?
[12]	Quantifying Maintenance Effort	Density of Smells	Maintenance Effort	4 Small Artifical Systems	Yes
[13]	Refactoring-Software Quality Relation	Existence of Refactoring	Weighted Methods per Class, Depth of Inheritence Tree, Number Of immediate Children subclasses etc.	3 Popular Open Source Object Oriented Libraries	No
[14]	Refactoring-Software Quality Relation	Existence of Refactoring	Dependency Oriented Complexity Metrics	3 Simple Software Design	No
This Thesis	Quantifying Maintenance Effort	Existence of Refactoring	Maintenance Effort	1 Live Project with nearly 300.000 LOC	Yes

CHAPTER 6

CONCLUSIONS

6.1 Summary of Conducted Work

For extracting the study results during the thesis implementation, a strict process was applied in a specific order. Since the project was big and the data was so dense, a path is followed strictly. The first analysis was started with the issue tracking system. Our aim was to find the issues that applied a refactoring technique for a code smell which obeys the previous studies of Martin Fowler and Joshua Kerievsky. After this process, the issues that have these characteristics were left. The next process was to analyze selected issues, because there were tens of classes that were modified under the specific issue; however, the aim of this issue was to solve a code smell that was related to the specific and problematic classes. Therefore, among the tens of classes, the elimination was conducted, and only the classes that were related to our context were selected. Then, the next process came up which was tracking the history of classes to find the spent efforts by programmers for fixing a defect or modifying the class for other refactoring need. This operation was done for the time interval both in the beginning of class and at the end of class. After finding issue frequency for different time intervals and issue types, average solution time of issues was calculated. The aim was to show that total programmers' effort was lower for the next time interval because of the implemented refactoring that was under research.

The main aim was to show the importance of refactoring concept for both the software itself and its effects on the programmers. The results of the study shows that after implementing predefined refactoring techniques for predefined code smells, the workload of the programmer that originates from refactoring and defect removing need, clearly decreases.

6.2 Discussion

In this thesis, some points should be emphasized for describing the challenging and open-ended points.

First of all, the calculated commits in the history of classes were only related with enhancement and defect issues. The reason is that an issue whose type is defect or enhancement means that there is a problem with the class. In other words defect and enhancement issues relate with the quality and structure of the class. However, other issues are not related with classes directly. They are related with the out of scope work for classes.

The next point is about the history of classes. In this thesis, class histories were analyzed for before and after the commit which is related with the issue under investigation. Moreover it was assumed that the improvement of effort was caused by that refactoring only. Although other commits and issues which were related with the class were also analyzed separately, there may be another issues that affected the improvement of this effort. However, it can be said that the number of these types of issues were limited, by looking at the data.

Another important point is about *"Large Class"* code smell. The history of classes that were refactored due to this code smell were analyzed in this thesis. However, the classes that were created during refactoring were not counted and their histories were ignored. Although this decision did not affect the results of my study significantly, including the created classes to this thesis would add to its outcome.

Finding code smell/refactoring technique pairs were another point which should be explained. In this thesis each selected pair was part of either Martin Fowler's proposed list or Joshua Kerievsky's proposed list. Although other code smells and related techniques exist, they were not included in this thesis.

6.3 Future Work

For future work, other projects and real world cases may be analyzed similar to this process. In this study the estimated workload could be defined in terms of total number issues and their average resolving time. Moreover resolving time of all issues were not available. Therefore

finding a bigger data set gives more accurate results and this point is important for future research.

Another important point is calculating the data for each specific code smell. In other words, in order to find the effort for maintaining a class during periods before and after a specific refactoring time, data of created new classes should be also calculated to obtain more reliable results.

Last important point is about analysis of normal classes. In this study, only issue frequencies of them were analyzed. For future work, average solution effort will be a valuable metrics to be included for the controlled part of the experiment.

REFERENCES

[1] R. France, S. Chosh, E. Song, and D.-K. Kim, "A metamodeling approach to pattern-based model refactoring," *Software, IEEE*, vol. 20, no. 5, pp. 52–58, 2003.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and don Roberts, *Refactoring: Improving the Design of Existing Code*. Object Technology International Inc., 2002.

[3] S. Counsell, R. Hierons, R. Najjar, G. Loizou, and Y. Hassoun, "The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph," in *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pp. 181–192, 2006.

[4] "Ejb." from http://en.wikipedia.org/wiki/Enterprise_JavaBeans, Dec. 2013. "last accessed on 30/12/2013".

[5] "Actionscript." from http://en.wikipedia.org/wiki/ActionScript, Dec. 2013. "last accessed on 30/12/2013".

[6] "Jira." from https://www.atlassian.com/software/jira, Dec. 2013. "last accessed on 30/12/2013".

[7] J. Kerievsky, *Refactoring to Patterns*. Boston, MA, USA: Pearson Education Inc., 2004.

[8] B. Walter and P. Martenka, "Looking for patterns in code bad smells relations," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp. 465–466, 2011.

[9] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Improving the precision of fowler's definitions of bad smells," in *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE*, pp. 161–166, 2008.

[10] C. L. Mika V. Mantyla, "Subjective evaluation of software evolvability using code smells: An empirical study," pp. 395–431, 2006.

[11] S. C. H. Hamza, T. Hall, and G. Loizou, "Code smell eradication and associated refactoring," pp. 102–107, 2008.

[12] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, 2013.

[13] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?," in *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, pp. 10–10, 2007.

[14] K. Reddy and A. Rao, "A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics," in *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on*, pp. 1011–1018, 2009.

[15] T. Sharma, "Quantifying quality of software design to measure the impact of refactoring," in *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pp. 266–271, 2012.

[16] "Smells to refactorings quick reference guide." http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf, Sept. 2005. "last accessed on 29/11/2013".

[17] "Subversion." from http://subversion.apache.org/, Dec. 2013. "last accessed on 30/12/2013".

[18] "Eclipse." from http://www.eclipse.org/, Dec. 2013. "last accessed on 30/12/2013".