DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DETECTING BAD SMELLS IN CODES BY USING ALGORITHM ANALYSIS

by

Aylin GÜZEL

August, 2016

İZMİR

DETECTING BAD SMELLS IN CODES BY USING ALGORITHM ANALYSIS

A Thesis Submitted to the

Graduate School of Natural and Applied Sciences of Dokuz Eylül University

In Partial Fulfillment of the Requirements for the Master of

Science in Computer Engineering

by

Aylin GÜZEL

August, 2016

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"DETECTING BAD SMELLS IN CODES BY USING ALGORITHM ANALYSIS"** completed by **AYLİN GÜZEL** under supervision of **ASST. PROF. DR. ÖZLEM AKTAŞ** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Özlem AKTAŞ

Supervisor

(Jury Member)

(Jury Member)

Prof.Dr. Ayşe OKUR
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisor Asst. Prof. Dr. Özlem AKTAŞ, for her support, supervision, constructive critisim, encouragement and useful suggestions throughout this study. It was a great honor to work with her for my thesis.

I would like to express my gratitude to all the academic staff of computer engineering department and the Graduate School of Natural and Applied Sciences.

I am also highly thankful to Prof. Alp KUT, Asst. Prof. Dr. Kökten Ulaş BİRANT, Asst. Prof. Dr. Özlem AKTAŞ, Asst. Prof. Dr. Canan Eren ATAY, Asst. Prof. Dr. Semih UTKU, Asst. Prof. Dr. Gökhan DALKILIÇ, Dr. Malik Kemal ŞİŞ and Dr. Melda DUMAN for their valuable suggestions and great lessons they provide during my education.

I would like to thank to the members of my jury Asst. Prof. Dr. Kökten Ulaş BİRANT, Asst. Prof. Deniz KILINÇ and Asst. Prof. Dr. Özlem AKTAŞ for their detailed and constructive comments.

I would like to offer my special thanks to my family for their support, patience, help and encouragement. I would not have been able to complete this thesis without their support and love.

Aylin GÜZEL

DETECTING BAD SMELLS IN CODES BY USING ALGORITHM ANALYSIS

ABSTRACT

Analysis, decision making, making design, detecting defects and correcting mistakes are important in software development process. Bad smell in code occurs in some cases, such as, wrong analysis, incorrect integration of new modules into the system, ignoring the software development principles, writing codes in complex way, designing system incorrectly etc. Bad smells reduce the quality of the software and an indication of potential problems in the system. Bad smells in the code must be destroyed for better quality, high-performance, low-cost, re-use, modification and easy development of software. Refactoring is simple but has a huge impact on software quality.

This work focuses on the definition of bad smell in codes, types of bad smell, occurence reasons, methods of eliminating code smells, when do we use Refactoring, Refactoring methods, Refactoring process, detecting bad smells in code by algorithm analysis approach and how the code could be done better.

In this thesis, some sorting algorithms periods were compared and their relationships with bad smells in code were explained. Additionally, the relationship between algorithm analysis and bad smells in code was examined. Performances of some sorting algorithms have been compared by using runtime calculations. Finally, in this thesis, comparison of the certain recursive and iterative sorting algorithms was made.

Keywords: Refactoring, software engineering, bad smells, algorithm analysis, code review, good code, code optimization.

ALGORİTMA ANALİZİ KULLANARAK KODLARDAKİ KÖTÜ KOKULARIN TESPİT EDİLMESİ

ÖZ

Yazılım geliştirme sürecinde analiz, karar verme, tasarım yapma kusurları tespit etme ve hataları düzeltme önemlidir. Kodlardaki kötü kokular yanlış analiz, yeni modüllerin sisteme yanlış entegre edilmesi, yazılım geliştirme prensiplerinin göz ardı edilmesi, karmaşık kodlar yazılması, sistemin yanlış tasarlanması gibi durumlarda meydana gelir. Kötü kokular yazılım kalitesini azaltır ve sistemdeki potansiyel problemlerin göstergesidir. Daha kaliteli, performansı yüksek, maliyeti düşük, başka bir yerde kullanılması, değiştirilmesi ve geliştirilmesi kolay yazılımlar için kodlardaki kötü kokuların yeniden düzenleme ile yok edilmesi gerekmektedir.

Bu çalışmada, kodlardaki kötü kokunun ne olduğu, kötü koku çeşitleri, oluşma nedenleri, kod kokularını yok etme yöntemleri, yeniden düzenlemeyi ne zaman kullanırız, yeniden düzenleme yöntemleri, yeniden düzenleme süreci, algoritma analizi yöntemi ile kodlardaki kötü kokunun tespit edilmesine ve nasıl daha iyi kod yazılabilineceğine odaklanılmıştır.

Tez çalışmasında, bazı sıralama algoritmalarının süresi karşılaştırılmış ve kodlardaki kötü kokuyla olan ilişkileri incelenmiştir. Ayrıca, algoritma analizi ve koddaki kötü kokuların arasındaki ilişki incelenmiş, bazı sıralama algoritmalarının performansları çalışma zamanı hesaplamaları kullanılarak kıyaslanmış, nihayetinde, bazı özyinelemeli ve tekrarlamalı sıralama algoritmalarının karşılaştırılması yapılmıştır.

Anahtar kelimeler: Yeniden düzenleme, yazılım mühendisliği, kötü koku, algoritma analizi, kod inceleme, iyi kod, kod iyileştirme.

CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER ONE
INTRODUCTION

1.1 General Information and Purpose

Code defects reduces the software quality. Code smells occurs in some cases: wrong analysis, thinking about the system wrongly, making the wrong decisions about system, ignoring the software development principles, writing codes in complex way, designing system incorrectly etc. This thesis describes code smells and solution of code smells in detail.

Bad smells are determined with the help of software engineers' point of view or using software tool. Also, code smells are destroyed manually or using the software code smell detection tool.

We aim at better software quality, high system performance, low-cost, re-use, modification and easy development of software. Thus, code smells must be destroyed by using Refactoring methods. Refactoring is the best solution for code smells.

This thesis focuses on identification and destruction of bad smells using software engineers' point of view. In this thesis, bad smells are destroyed manually. The relationship between algorithm analysis and bad smells in code is examined. Performances of some sorting algorithms are compared by using runtime calculations. In addition, in this thesis, how to write better code is examined in detail.

1.2 Organization of the Thesis

This thesis includes eight chapters and the rest of this thesis is organized as follows:

In Chapter 2, general information about related works and literature search about code smells, Refactoring and code optimization.

In Chapter 3, “definition of code smell”, “code smell occurrence reasons” and “solution of code smells” have been explained in detail.

In Chapter 4, some code smells are described generally.

In Chapter 5 gives information about Refactoring, Refactoring Process, “benefits obtained from Refactoring” in detail.

In Chapter 6, some Refactoring methods are described in detail.

In Chapter 7, bad smells in code and Algorithm Analysis approach is examined in detail.

Finally, in Chapter 8, the conclusion remarks and future works have been given.

CHAPTER TWO

RELATED WORKS

2.1 Literature Review

Mens and Tourwe´ (2004) have implemented refactoring methods in a code that are required for the current structure. They make the design more clear, easy to intervention, suitable for object-oriented design principles. In this study, the design has been optimized by using refactoring methods for wrong designed Document Class and by explaining through the sample. The wrong design example used in this study is shown in Figure 2.1.

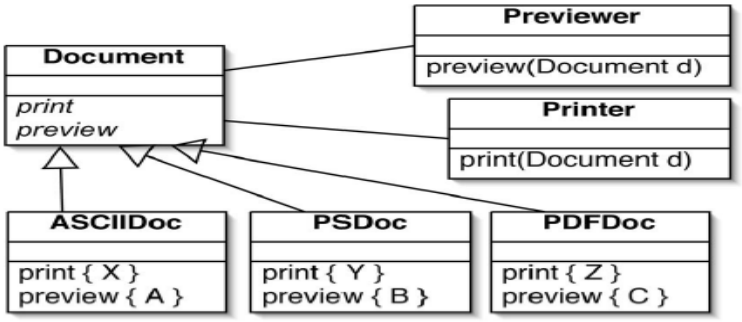

Figure 2.1 Document class hierarchy and helper classes.

This design is not optimal because different functionalities of the Document class are distributed over all the subclasses. In order to add a new functionality to the Document class, such as a text search or a spell checker, we need to change every subclass of Document and we need to define the appropriate helper classes.

To overcome these problems, the design needs to be refactored. By adding Visitor Class in design is ensured to incorporate all subclasses. Required method and the variable name changes, the carriage of necessary methods in appropriate places, the design have been optimized using the basic refactoring methods such as adding a new class. The optimized design by using refactoring method is shown in Figure 2.2

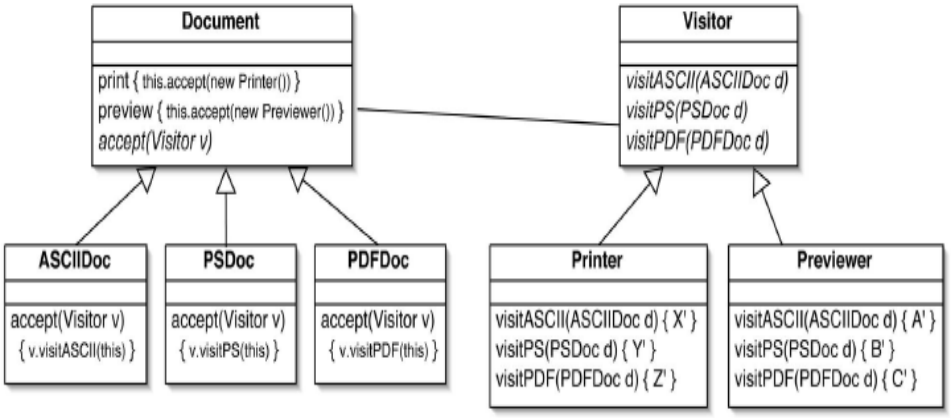

Figure 2.2 Refactored design model for the document class hierarchy.

Liu and his friends emphasized that why and when software should be refactored. The tools are expected to detect bad smells automatically or semiautomatically. Most bad smells automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools. It is up to software engineers to determine how to restructure bad smells in terms of refactoring rules that should be applied. Not all refactorings are supported by refactoring tools. As a result, detecting and resolving bad smells remain time-consuming, even with tool support (Liu, Zhiyi, Shao & Niu, 2012). Detection and resolution of bad smells is shown in Figure 2.3.

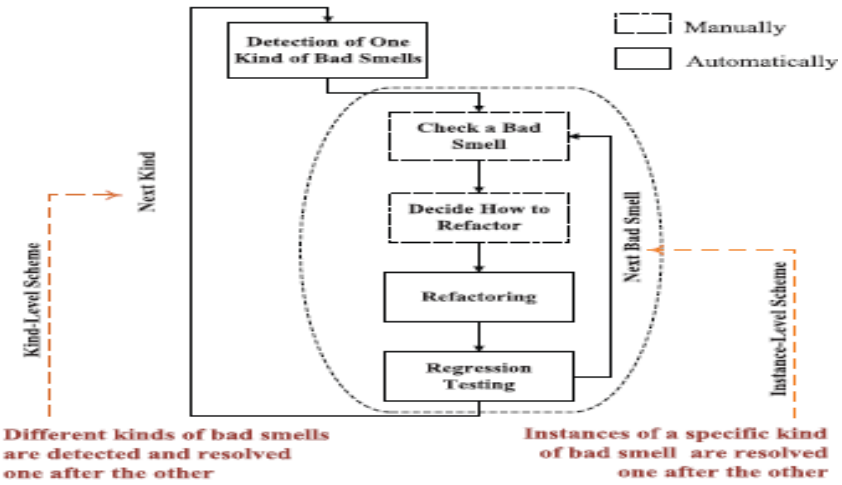

Figure 2.3 Detection and resolution of bad smells.

Chatzigeorgiou and Manakos (2014) emphasized that the design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software. They have detected four bad smell using JDeodarant Tool on a valid Java code. Methods suffering from the Long Method code smell are usually pieces of code with large size, high complexity and low cohesion which consequently require more time and effort for comprehension, debugging, testing and maintenance. This problem is solved by using automatic tool support or simplifies the code by breaking large methods into smaller ones.To identify large and complex classes ("God" Class) JDeodarant Tool "Clustering Algorithm" approach was used.

Khomh and his hriends stated that researchers and practitioners had developed a variety of approaches to detect bad smells in the code and design so far, however; these approaches can not solve the stated uncertainty in the process of detecting the bad smell. A "Bayesian Approach" (bbns) is used to detect bad smells in the code. The approach shown on “Blob AntiPattern”. BBN has been evaluated on two test programs and was observed to be successful (Khomh, Vaucher, Gueheneuc & Sahraoui, 2009).

Moha (2007) stated that design defects come from poor design choices and have the effect of degrading the quality of object-oriented designs. Also, he stated his research design defects have not been precisely specified and there are few appropriate tools that allow their detection as well as their correction. His goal is to provide a systematic method to specify systematically design defects precisely and to generate automatically detection and correction algorithms from their specifications. To overcome the problems stated previously, he propose a method, called DECOR (Defect dEtection for CORrection), to specify systematically high-level design defects and to generate detection and correction algorithms from their specifications semi-automatically. DECOR, is based on description of the design defects, detect, correction and verification respectively.

Malhotra and Pritam (2012) in this study the authors attempt to empirically validate whether it is possible to determine the degree of change proneness for a

class on the basis of certain code smells in an object- oriented system. The data used for assesment are the source code of Quartz, an open source job schedular. A total of 79 classes are examined in this study. The result suggest a clear relationship between code smells and change proneness of a class. Also, a tool has been developed using thresholds to identify 13 different bad smell in the Java class.

Schumacher et al. (2010) investigated that a wide range of classes that cause bad smell in the code ("God" Class) how is recognized by professional software developers and studied how they bring solutions to this class. For solution, these classes split up into multiple classes, or else sub-classes should be extracted from the god class. Metric-based approach is used for the recognition of the huge class by using CodeVizard software tool. This component is used to parse C # program and to calculate code metrics. In this research proved that both people and software tool can identify the very large classes.

.

Kessent and his friends which emerged inspired by Darwin's theory of evolution in their study using genetic algorithm to improve the bad smell in the code. The results reported of an evaluation of their approach using four open-source projects. Their proposal achieved high correction scores by fixing the majority of expected bad smells (Kessentini, Mahaouachi & Ghedira, 2013). General approach used for detecting bad smells is shown in Figure 2.4.

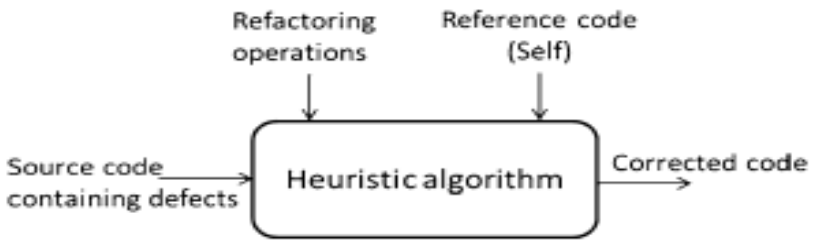

Figure 2.4 General architecture of the approach.

Rech and Schäfer (2007) have developed the CodeSonar tool to improve the quality of software systems. CodeSonar was developed to support software engineers during software development and maintenance activities through the visualization of

source code and quality defects using a visual interface in the eclipse IDE. In this study, software testing, software product metrics, software visualization were used Refactoring Tecniques for the discovery of defects in the code and visualization for software control again.

Sjøberg and his friends emphasized that code smells indicate bad design that leads to less maintainable code. Also, their work investigates the relationship between code smells and maintenance effort. This study was conducted on four different but functionally equivalent (with the same requirements specifications) web-based information systems originally implemented in Java by different six developers from different companies. Each developer spent three to four weeks and totally, they modified 298 Java files in the four systems.

An Eclipse IDE plug-in measured the exact amount of time a developer spent maintaining each file. Regression analysis was used to explain the effort using file properties, including the number of smells. None of the 12 investigated smells was significantly associated with increased effort after they adjusted for file size and the number of changes. File size and the number of changes explained almost all of the modeled variation in effort. The effects of the 12 smells on maintenance effort were limited. To reduce maintenance effort, a focus on reducing code size and the work practices that limit the number of changes may be more beneficial than refactoring code smells (Sjøberg, Yamashita, Anda, Mockus & Dyba, 2013).

System	A		B		C		D		Total	
Number of Java files	63		168		29		119		379	
Java LOC	8205 LOC		26679 LOC		4983 LOC		9960 LOC		49827 LOC	
Code smell	**N**	**Density**	**N**	**Density**	**N**	**Density**	**N**	**Density**	**N**	**Density**
Feature Envy	37	4.51	34	1.27	17	3.41	25	2.51	113	2.27
Data Class	12	1.46	32	1.20	9	1.81	24	2.41	77	1.55
Temporary variable used for several purposes	12	1.46	31	1.16	6	1.20	4	0.40	53	1.06
Shotgun Surgery	7	0.85	17	0.64	0	0.00	13	1.31	37	0.74
ISP Violation	7	0.85	8	0.30	1	0.20	11	1.10	27	0.54
God Method	4	0.49	14	0.52	3	0.60	5	0.50	26	0.52
Refused Bequest	17	2.07	8	0.30	0	0.00	1	0.10	26	0.52
Data Clump	8	0.98	2	0.07	3	0.60	8	0.80	21	0.42
God Class	1	0.12	5	0.19	3	0.60	2	0.20	11	0.22
Duplicated code in conditional branches	1	0.12	4	0.15	2	0.40	2	0.20	9	0.18
Implementation used instead of interface	5	0.61	4	0.15	0	0.00	0	0.00	9	0.18
Misplaced Class	0	0.00	2	0.07	0	0.00	2	0.20	4	0.08

Figure 2.5 Number and density of code smells per type in the systems.

Rahman et al. (2012) have investigated whether code cloning is really a “bad smell” (Fowler, Beck, Brant, Opdyke & Roberts, 1999) by using several medium to large projects. Generally known that Code Clones increases project size and maintenance costs. In their work, they try to validate whether cloning makes code more defect prone. This paper analyses the relationship between cloning and defect proneness. Consequently, the great majority of bugs are not significantly associated with clones. Also, they find that code clones may be less defect prone than non-cloned code and other findings about coloning. Their work do not support the claim that clones are really a “bad smell”.

Murphy-Hill et al. (2012) investigated refactoring tool usage and evaluate some of the assumptions made by other researchers. To measure tool usage, they randomly sampled code changes from four Eclipse and eight Mylyn developers and ascertained, for each refactoring, if it was performed manually or with tool support. They found that refactoring tools are seldom used. They cast doubt on several previously stated assumptions about how programmers refactor, while validating others by using their special data. Also, the investigors of this research interviewed the Eclipse and Mylyn developers to understand why they did not use refactoring tools and to gather ideas for future research.

Buschmann (2011) emphases that Refactoring improves the quality of some part of a system while preserving its functional behavior. Also, Refactoring isn’t limited to code detail but can range up to the larger scale of a system’s software architecture. He told a real story in his paper for supporting these ideas by talking with a project team about Refactoring. In addition, Refactoring, maintain a system’s high developmental quality. If regularly practiced, refactoring has a positive effect on developer habitability and system life cycle costs. Consequently, this paper’s author defends that refactoring meets reengineering and rewriting, two other common approaches for improving system quality.

Liu et al. (2014) illustrated that Refactorings might be done using two different tactics which are root canal refactoring and floss refactoring. Root canal refactoring is to set aside an extended period specially for refactoring. Floss refactoring is to

interleave refactorings with other programming tasks. The authors carry out a case study to analyse the usage data information collected by Eclipse usage data collector. Results suggest that about 14% of refactorings are root canal refactorings. These findings reconfirm the hypothesis that, in general, floss refactoring is more common than root canal refactoring.

	Root canal refactoring	Floss refactoring
extended period for pure refactorings	yes	no
refactoring scope	large (e.g. the whole application)	small (e.g. a newly modified class)
smell detection tools	needed	not needed
number of refactoring opportunities found at a time	large	small
schedule algorithm	needed	not needed
interweaved with non-refactoring activities	no or slightly	heavily
seamless switch between refactoring and routine development	not needed	needed

Figure 2.6 Difference between refactoring tactics.

Mantyla and Lassenius (2006) propose use of the term software evolvability (software maintainability) to describe the ease of further developing a piece of software and outline the research area based on four different viewpoints. Furthermore, they describe the differences between human evaluations and automatic program analysis based on software evolvability metrics. They suggest that organizations should make decisions regarding software evolvability improvement based on a combination of subjective evaluations and code metrics.

Li and Shatnawi (2007) aim at find empirical evidence of the association between the bad smells and class error probability by using the error data from the three releases of the Eclipse Project. They found that the Shotgun Surgery, God Class and God Methods bad smells were positively associated with the class error probability. These results provided the first empirical evidence that some bad smells can indeed indicate class error probability in an object-oriented system as the system's design continues to evolve after its official release. The finding also suggests that

refactoring a class, besides improving the architectural quality, reduces the probability of the class having errors in the future.

Yamashita (2014) investigates the capability of twelve code smells to reflect actual maintenance problems. For this purpose, four medium-sized systems with equivalent functionality but dissimilar design were examined for code smells and also six software developers worked for his research. During that period, researcher recorded problems faced by developers and the associated Java files on a daily basis. He developed a binary logistic regression model, with “problematic file. Twelve code smells, file size, and churn constituted the independent variables. As a result, code with ISP violation should be considered potentially problematic and be prioritized for refactoring.

Stroustrup (1998) wanted to show his view of what object-oriented means in programming languages. For this purpose, the author presented examples in C++ because C++ is one of the few languages that supports data abstraction, traditional programming techniques and object-oriented programming, The author supported that object–oriented programming, thinking and using must be as a lifestyle. Also, the author of this paper mentioned that advanteges of object-oriented programming, effectively using, the best features of object-oriented programmig with an easy to understand examples.

Dyke and Kunz (1989) mentioned about Object-Oriented Programming and its advantages, what is Smalltalk, how do we use Smalltalk, the logic of Smalltalk, structural characteristics of Smalltalk. For instance, In Smalltalk, all data elements, including integers are considered as objects that have associated methods. For instance, In Smalltalk, the operation “2+3” is interpreted as sending the “+” message with the argument “3” to the integer object “2”. The integer class has a method “+” that is inherited by the instance object, “2”, which performs the addition and returns the sum “5” to the sender. There is a class hierarchy (object class) in Smalltalk. For

instance, integer class is a subclass of Number. In short, everything in Smalltalk is implemented as an object.

Garrido and Rosst (2011) mentioned that history of Refactoring, Refactoring process, what is Refactoring, when do we use Refactoring with an Web Applications, classification of Web refactorings, results and impact of Refactoring, example of Web Application Refactorings.

Refactoring	Intent	Scope
Convert images to text[5] In webpages, replace any images that contain text with the text they contain, along with the markup and CSS rules that mimic the styling.	Accessibility	Code
Add link[7,11] Shorten the navigation path between two nodes.	Navigability	Navigation model
Turn on autocomplete[5] Save users from wasting time in retyping repetitive content. This is especially helpful to physically impaired users.	Effectiveness, accessibility	Code
Replace unsafe GET with POST[5] Avoid unsafe operations, such as confirming a subscription or placing an order without explicit user request and consent, by performing them only via POST.	Credibility	Code
Allow category changes[7] Add widgets that let users navigate to an item's related subcategories in a separate hierarchy of a hierarchical content organization.	Customization	Presentation model
Provide breadcrumbs[7] Help users keep track of their navigation path up to the current page.	Learnability	Presentation model

Figure 2.7 Classification of web refactorings.

Wirfs-Brock (2008) mentioned that Jon Bentley's thesis and Kent Beck in Implementation Patterns (Addison-Wesley, 2007) study. Also, she emphasized that importance of good design, good programming, writing beautiful code, importance of consistent code, test- driven development, importance of writing easy to understand, easy to read codes, importance of purify that unnecessary complexity of codes and design, importance of documentation. For instance, "Do not believe any programmer, manager, or salesperson who claims that code can be self-documenting or automatically documented" ("Comments are More Important than Code," ACM Queue, Sept./Oct. 2007).

Booch (2014) mentioned that computing, computer science history with examples, Computational thinking, designing systems, implementation, solving problems, understanding human behavior by drawing on the concepts fundamental to computer science, learning to code, teaching coding spaces, importance of coding.

Devarakonda (1998) mentioned that active research and innovation in Object-Oriented Programming and Object- Oriented systems, Small-Talk, Corba, ActiveX, C++, and Java Technologies, extensibility for Object-Oriented Systems, reusability for Object-Oriented Systems, distrubuted programs, distrubuted objects and also performance of all.

Kim and Notkin (2005) studied that programmers often create similar code or reuse existing code by copying and pasting. It cause problems during software maintenance because programmers may need to locate code clones and change them consistently. In this work, they investigate how code clones evolve, how many code clones impose maintenance challenges, and what kind of tool or engineering process would be useful for maintaining code clones. Based on a formal denition of clone evolution, they built a clone genealogy tool that automatically extracts the history of code clones from a source code repository (CVS). Their clone genealogy tool enables several analyses that reveal evolutionary characteristics of code clones. Their initial results suggest that excessive refactoring may not be the best solution for all code clones; thus, they propose alternative tool solutions that assist in maintaining code clones using clone genealogy information.

Thompson and Li (2013) mentioned what is Refactoring for functional programming languages first in theory, and then in the context of a larger example. They identified Refactoring is the process of changing the design of a program without changing what it does like that.

This paper reflects on their experience of building tools to refactor functional programs. They discuss various extensions to the core tools, including integrating the tools with test frameworks; facilities for detecting and eliminating code clones; and

facilities to make the systems extensible by users. They have presented an overview of refactoring functional programs, and shown how different the process and its implementation can be for two representative functional for another perspective on this from the C verification community. A number of refactorings that they have implemented particularly the structural ones are similar to Object-Oriented Refactorings; other Object-Oriented refactorings which move methods and attributes around the inheritance hierarchy do not have direct equivalents in the functional paradigm. Their work illustrated has underlined that it is important not only to implement the basic refactorings but also to provide decision support tools, such as clone detection and module analysis, that can guide the application of the tool.

CHAPTER THREE
BAD SMELLS IN CODE

3.1 Definition of Code Smell and Occurrence Reasons

Design anomalies in the codes are also called bad smells. Code smells are signs of potential problems in the system. Bad smells in the code reduce the quality of the software. Wrong analysis, bad design, incorrect integration new modules into the system, ignoring the software development principles, writing codes in complex way, thinking about the system incorrectly, making wrong decision about the system, writing codes with poor readability and understandability for only to recover the moment etc. causes bad smells (Güzel & Aktaş, 2015).

3.2 What We Do When Code Smells Occur in the Code ?

Software engineers should be written the codes adhering to the software development process and principles. Design, analysis, testing, review, maintenance and workflow problems may ocur in the case of ignoring any stage of the process or principles. At this point, code smells must be destroyed manually or using tool support. This thesis uses only manual resolving method for code smells problem.

Detecting and resolving bad smells is time-consuming for software engineers despite proposals on code smell detection and refactoring tools. Number of code smells are increasing continuously, yet tool support for detecting bad smells is not enough. Therefore, software engineer’s point of view is used for detecting and correcting bad smells in codes commonly.

Software engineers should be given the right decisions about the system also codes and should be aimed to improve software performance, readability, flexibility, understandability. For this purpose, the required procedure is called Refactoring.

Bad smells in the code must be destroyed for better quality, high-performance, low-cost, elsewhere use, modification and easy re-configuration. Refactoring modifies software to improve its readability, maintainability, and extensibility without changing what it actually does. Refactoring does not alter the external behavior of the code, yet improves its internal structure. The goal of Refactoring is simply not adding any new functionalities in the software. Refactoring is simple but has an enormous impact on software quality.

Bad smell in the code causes a loss of performance in the software. Loss of performance would give rise to increased costs, the motivation disorders; transportability, interchangeability and readability of code that are difficult. Refactoring should be used to improve the performance of the software.

CHAPTER FOUR

CODE SMELL SAMPLES

In this chapter, some important bad smells are examined (Güzel & Aktaş, 2016). Code smells are detected with the help of a software engineer's point of view or the software tool. Also, code smells can be solved manually or using the software tool.

4.1 Duplicated Code

Duplicated code is a really big problem in terms of having good design. "Duplicated code can be defined as a same code structure in more than one place in application or code." (Fowler, Beck, Brant, Opdyke & Roberts, 2002). The program will be better by unifying the code smells in an efficiently way.

The duplicated code divided into some categories: same expression in two methods of the same class, the same expression in two identical subclasses etc. The first one can be corrected by using Extract Method and the code can be called from both places. Second one can be eliminated using Extract Method in both classes then Pull Up Field.

4.2 Long Method

The best programs are written by using not long methods. Short methods are important for good Refactoring. The longer procedure makes more difficult to understand the code so, causes the bad smells. Therefore, the methods can be shorten by using Extract Method approach (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.3 Large Class

Large Class means a class with a huge amount of variables. In other words, large class means that a class is trying to do too much unexpected tasks without necessary. Solution of large class problem is to eliminate redundancy in the class itself by using Extract Class or Extract Subclass approach (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.4 Long Parameter List

Parameters should be used only if necessary. Also, using global data is not optimal for programs. Global data was alternative to parameters. Instead of global data or large parameter lists, use objects. Use small parameter lists with an object-oriented programs. Short parameter lists are easy to understand and useage. “If the parameter list is too long or changes too often, rethink dependency structure of code and redesign your own code” (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.5 Divergent Change

“Divergent change occurs when one class is commonly changed in different ways for different reasons. Thus, each object is changed only as a result of one kind of change” (Fowler, Beck, Brant, Opdyke & Roberts, 2002). For solution, use Extract Class method.

4.6 Shotgun Surgery

"Shotgun surgery is similar to divergent change, yet is the opposite".Make a lot of little changes to a lot of different classes use Move Method and Move Field to put all the changes into a single class for optimal solution. Inline Class approach is used for to bring a whole bunch of behavior together. "Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes" (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.7 Data Class

Data classes have fields and getting and setting methods for the fields. Data classes are dumb data keepers. If getting and setting methods used by other classes, try to use Move Method to move behavior into the data class. If you can't move a whole method, use Extract Method to create a method that can be moved. Data classes are approved as a starting point, but to participate as a grownup object, they need to take some responsibility (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.8 Switch Statements

The most common problem of switch statements is duplication. In other words, the same switch statement is found more than one places in the program. If you add a new clause to the switch, you have to find all these switch, statements and change them (Fowler, Beck, Brant, Opdyke & Roberts, 2002).

4.9 Comments

Refactoring not to say that people shouldn't write comments. "In early stage, comments aren't a bad smell; indeed they are a sweet smell" (Fowler, Beck, Brant,

Opdyke & Roberts, 2002). But now, supported that commented code used for hiding the bad smells and so the comments are bad smell, because the code is bad. If our code is good enough for reading, re-using, implementing to other systems, there is no need to write any comment into code.

4.10 Lazy Class

Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted. Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small or perhaps it was designed to support future development work that never got done. For subclasses with few functions, try Collapse Hierarchy. Advantage of this solution is reduced code size and easier maintenance. Ignore this approach when a Lazy Class is created in order to delineate intentions for future development, try to maintain a balance between clarity and simplicity in your code (Lazy Class, n.d.).

CHAPTER FIVE
REFACTORING

5.1 What is Refactoring?

Refactoring aims to improve software performance, readability, flexibility and understandability. Refactoring does not change the external behavior of the code. In addition, Refactoring improves internal structure of the code. The goal of refactoring is not to add new functionality. In other words, Refactoring does not change the observable behavior of the code. Refactoring is used for destruction of code smells.

5.2 When Do We Use Refactoring?

Refactoring is used for better quality, high-performance, low-cost, elsewhere use, modification and easy re-configuration. Refactoring is used some cases as follows:

- when a new function is added,
- if the existing design and code ‘bad’,
- to correct errors,
- code reviewing.

5.3 Benefits Obtained from Refactoring

Refactoring builds up new hierarchies. Refactoring does not change the observable behavior of the code, yet improves its internal structure. Refactoring provides some advantages for system:

- simple and understandable,
- easy to change,
- readability is high,

- used in other projects.

Benefits obtained from Refactoring are as follows:

- code quantity is reduced.
- complex code is simplified.
- code maintenance is facilitated.

5.4 Refactoring Process

Refactoring targets clean, good, simple, understandable and high readable code. Refactoring process is as follows:

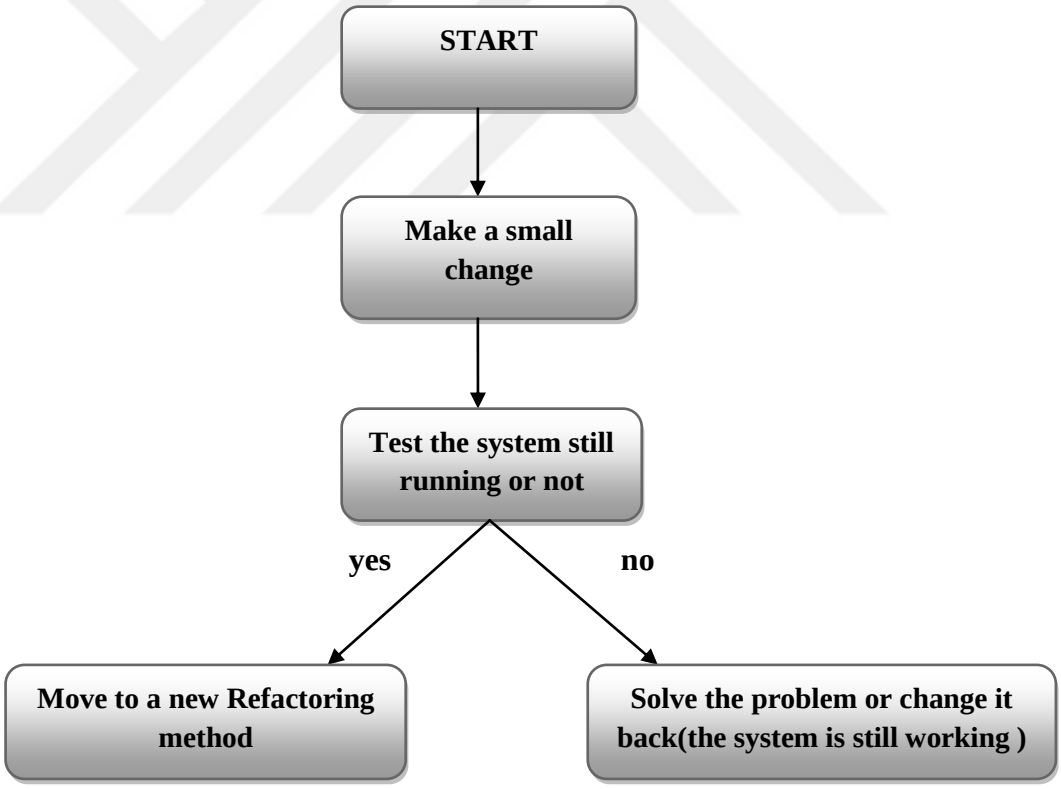

Figure 5.1 Refactoring process.

CHAPTER SIX
SOME REFACTORING METHODS

Refactoring methods are basically divided into six subcatagories as follows: Composing Methods, Moving Features Between Objects, Organizing Data, Simplifying Conditional Expressions, Making Method Calls Simpler, Dealing with Generalization (Güzel & Aktaş, 2016).

6.1 Composing Methods

Some Composing Methods are: Extract Method, Inline Method, Inline Temp, Split Temporary Variable, etc (Composing Methods, n.d.).

6.1.1 Extract Method

Extract method is generally used in order to simplify the long methods. In this example, the simple code fragment that collects four number will be restructured respectively. The bad code is shown in Figure 6.1.

```
namespace Extract_Method_Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables for adding process
            int a;
            int b;
            int c;
            int d;

            // Enter the values for addition
            Console.Write("Write a:");
            a = Convert.ToInt32(Console.ReadLine());
            Console.Write("Write b:");
            b = Convert.ToInt32(Console.ReadLine());
            Console.Write("Write c:");
            c = Convert.ToInt32(Console.ReadLine());
            Console.Write("Write d:");
            d = Convert.ToInt32(Console.ReadLine());
            int e = a + b + c + d;
            Console.WriteLine("e:{0} ", e); // Result value for additon
            Console.ReadKey();
        } } }
```

Figure 6.1 The bad code.

In Figure 6.1, bad smells such as dublication, naming failure for variables (int a, int b, int c, int d, int e), error for adding comment lines, square brackets design error, more lines are observed. This code is not optimal in terms of Refactoring. This bad code is not readable, clean and simple. So, begin the Refactoring process immediately.

First, square brackets design error corrected. Then, naming failure for variables and others are corrected. Unneccessary comment lines are cleaned. Refactoring defends using very little comment lines. Because Refactoring supports that the desired task should be understood from the written code, not from the comment lines. If your code is the best, you don't need to add any comment lines. Instead of writing comment lines, write the best code!

In Figure 6.1 "write a", "write b" or "e:{0}" etc. lines are not optimal. Because the variables (a,b,c,d or e) are numerical value or string value is not obvious. So, the program will crash when entering the values. Also, we don't know "What is e ?". Only, code writer knows its mean. Thus, bad smell is available.

In addition, the variable declaration is corrected. More lines observed because of variable declaration failure. The integer variable declaration is done one by one. Thus, code lines are reduced. In additon, bad smell is destructed and algorithmic time is reduced. Now, the more readable code is shown in Figure 6.2.

```
namespace Extract_Method_Sample
{
    class Program
    {
        static void Main(string[] args)
        {

            int num1,num2, num3, num4;

            Console.Write("Enter Number:");
            num1 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num2 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num3 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num4 = Convert.ToInt32(Console.ReadLine());
            int sum = num1 + num2 + num3 + num4;
            Console.WriteLine("Sum:{0} ", sum);
            Console.ReadKey();
        }
    }
}
```

Figure 6.2 The better code.

Other code smell is available currently. All code is in main structure and code's readability is very poor. So, Extract method is used for correcting this smell. Extract method Refactoring path is shown in Figure 6.3 and Figure 6.4.

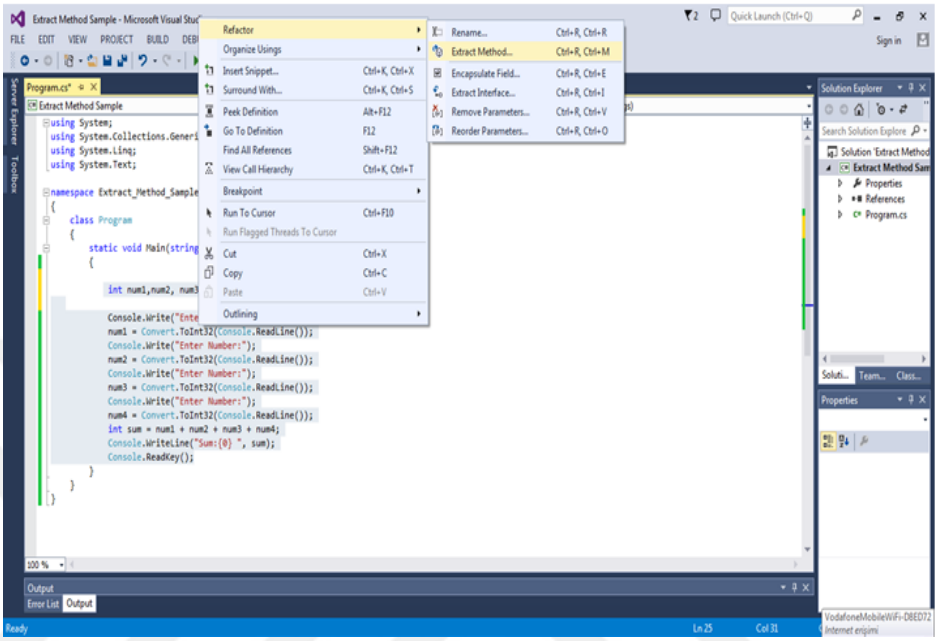

Figure 6.3 The extract method refactoring path.

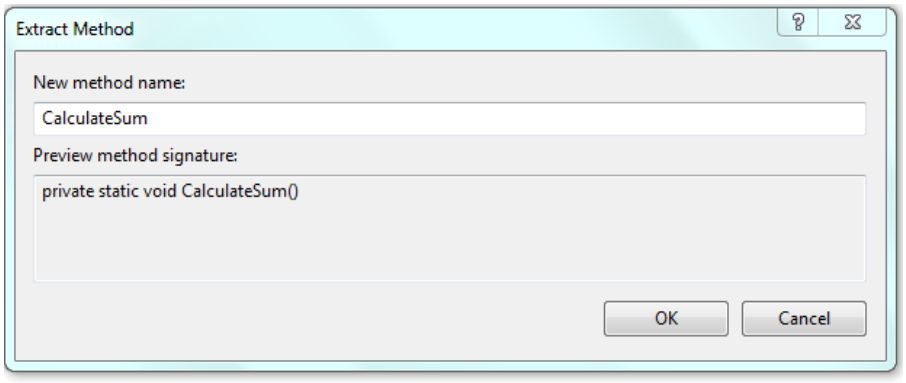

Figure 6.4 The extract method naming.

The new Refactoring is shown in Figure 6.5.

```
namespace Extract_Method_Sample
{
    class Program
    {
        static void Main(string[] args)
        {

            CalculateSum();
        }

        private static void CalculateSum()
        {
            int num1, num2, num3, num4;

            Console.Write("Enter Number:");
            num1 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num2 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num3 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter Number:");
            num4 = Convert.ToInt32(Console.ReadLine());
            int sum = num1 + num2 + num3 + num4;
            Console.WriteLine("Sum:{0} ", sum);
            Console.ReadKey();
        }
    }
}
```

Figure 6.5 The extract method refactoring.

After that, for loop is used for more optimal solution instead of other refactoring methods. Better solution is shown in Figure 6.6.

```
namespace Extract_Method_Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            int endVar, i, result = 0;

            Console.Write("Enter end border number : ");
            endVar = int.Parse(Console.ReadLine());
            for(i=0; i < endVar;i++)
            {
                Console.Write("Enter number : ");
                int num = int.Parse(Console.ReadLine());

                result = result + num;

            }

            Console.Write("Sum of numbers: {0}", result);
            Console.ReadKey();
        }

    }
}
```

Figure 6.6 The for loop solution for better code.

Finally, extract method is applied to the loop solution. The best result of Refactoring process is shown in Figure 6.7.

```
namespace Extract_Method_Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            CalculateSum();
        }

        private static void CalculateSum()
        {
            int endVar, i, result = 0;

            Console.Write("Enter end border number : ");
            endVar = int.Parse(Console.ReadLine());
            for (i = 0; i < endVar; i++)
            {
                Console.Write("Enter number : ");
                int num = int.Parse(Console.ReadLine());

                result = result + num;

            }

            Console.Write("Sum of numbers: {0}", result);
            Console.ReadKey();
        }

    }
}
```

Figure 6.7 The for loop solution with extract method.

As a result, the sample code is more readable and more useful than the start of the event. Refactoring process is simple, yet it has a immense impact on software.

6.1.2 Inline Method

Inline method aims at clean and simple code by reducing the number of unneeded methods. Inline method Refactoring process is (Inline Method, n.d.) :

- First, make sure that the method is not redefined in subclasses. If the method is redefined, refrain from this technique.
- Find all calls to the method. Replace these calls with the content of the method.
- Delete the method.

The code that needs to improve by Refactoring technique called Inline Method is shown in Figure 6.8 (Inline Method, n.d.) :

```
    void WriteMessage(String Message){
    Console.Write(Message);
    }
    void TestMethod(){
    WriteMessage("Inline Method Refactoring");
}
```

Figure 6.8 Source code.

The code which applied Inline Method Refactoring is shown in Figure 6:

```
    void TestMethod(){
      Console.Write("Inline Method Refactoring");
    }
```

Figure 6.9 Inline method refactoring.

6.1.3 Inline Temp

Inline temp improve the readability of the code by do away with the unnecessary variable. Temporary variable is only used once. Refactoring process of inline temp is (Inline Temp, n.d.):

- find all places that use the variable. Instead of the variable, use the expression that had been assigned to it.
- delete the declaration of the variable and its assignment line.

The code block which needs to Refactor is shown in Figure 6.10.

```
    double area (x, y)
    double x,y;
{
    double s;
    s=x*y;
    return s;
}
```

Figure 6.10 The bad function example.

The Inline Temp refactored code is shown in Figure 6.11.

```
   double area (x, y)
   double x,y;
{
   return x*y;
}
```

Figure 6.11 Inline temp refactored code.

Lastly, naming failure is corrected shown in Figure 6.12.

```
   double area (short_edge, long_edge)
   double short_edge,long_edge;
{
   return short_edge*long_edge;
}
```

Figure 6.12 Refactored code.

6.1.4 Split Temporary Variable

When you have the same temporary variable assigned to more than once, split it up into two, unless it is a loop variable (Composing Methods (Split Temporary Variable), n.d). Refactoring process of Split Temporary Variable is (Split Temporary Variable, n.d) :

- find the variables in the code assigned to more than once.
- use the new name instead of the old one.
- repeat this until the variable is assigned a different value.

The bad code design in Figure 6.13.

```
double temp = 2 * (a + b + c + d);
System.out.println(temp);

temp = ((a + b) /2) * h;
System.out.println(temp);
```

Figure 6.13 The bad code example.

The Split Temporary Variable refactored code is shown in Figure 6.14.

```
readonly double perimeter_trapezoid = (lower_base + upper_base + parallel_edge1 + parallel_edge2);
System.out.print(perimeter_trapezoid);

readonly double area_trapezoid = ((lower_base + upper_base) /2) * height ;
System.out.print(area_trapezoid);
```

Figure 6.14 Refactored code.

6.2 Moving Features Between Objects Methods

Some “Moving Features Between Objects” methods are: Move Method, Move Field, Extract Class, Hide Delegate, Remove Middle Man, etc (Moving Features Between Objects, n.d).

6.2.1 Move Method

A method is used more in another class than defined class is called Move Method Refactoring problem. Solution for move method problem is (Move Method, n.d.) :

- create a new method in the class that uses the method the most.
- move code from the old method to there.
- turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

6.2.2 Move Field

Move method problem is described as a field is used more in another class than defined class. Solution of move field problem is (Move Field, n.d.):

- create a new field in a new class
- change all its users (redirect all users of the old field to new one.)

6.2.3 Extract Class

One class does the work of two known as a Extract Class problem. This method replaces a single class that is responsible for multiple tasks with several classes each having a single responsibility.

6.2.4 Hide Delegate

Hide delegate defined as a client is calling a delegate class of an object. Solution of Hide Delagete problem is (Fowler, Beck, Brant, Opdyke & Roberts, 2002):

- create methods on the server to hide the delegate.

6.2.5 Remove Middle Man

A class has too many methods that simply delegate to other objects problem is called Remove Middle Man problem. This problem can be solved by deleting these methods and calling the end methods directly.

6.3 Organizing Data Methods

Some “Organizing Data” methods are: Encapsulate Collection, Encapsulate Field, Replace Array with Object, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, etc.

6.3.1 Encapsulate Collection

Collection means arrays, lists or vectors in object – oriented programming. A class contains a field. Also, a field contains a collection of objects. Working with the collection provided by using getter and setter methods. The problem is method’s returning value. A method returns a collection. For the best solution of this problem:

- Make method return a read-only view or,
- Create a add/remove methods.

Benefits of this Refactoring method is:

- The collection field is encapsulated inside a class. This method prevents accidental changing or overwriting of the collection elements by returning a copy of collection when the getter and seter method is called.

6.3.2 Encapsulate Field

Encapsulation is important for object-oriented programming and design. Using public field has some disadvantages such as data can change without any control. Thus, Encapsulate Field problem can be solved:

- First, use private field access.
- Then, create access methods (getting and setting methods).

Encapsulate field problem is shown in Figure 6.15.

```
class Client{
public string name;
}
```

Figure 6.15 Bad practice for encapsulation.

```
class Client{
private string name;
public string Name
{
get { return name; }
set { name = value; }
}
}
```

Figure 6.16 Refactored code block.

6.3.3 Replace Array with Object

Arrays contains various types of data. When we use arrays, some problems may ocur such as accessing problem to the array elements. Data stored place (array cells) maybe selected wrong. So, it causes bad smelss and accessing problem. The code smell example shown in Figure 6.17.

```
string[] book_info;
book_info=new string[3];

book_info[0] = "Introduction to Programming";
book_info[1] = "584";
book_info[2] = "75YTL";
```

Figure 6.17 Bad practice for encapsulation.

Refactored code with Replace Array with Object method shown in Figure 6.18.

```
  static void Main(string[] args)
{
  BookStore book_info = new BookStore();
  book_info.BookName="Introduction to Programming";
  book_info.BookPage="584";
  book_info.BookPrice="75";
}

  public class BookStore{

  public string BookName;
  public string BookPage;
  public string BookPrice;

}
```

Figure 6.18 Refactored code block.

6.3.4 Change Unidirectional Association to Bidirectional

This problem is described: the association between two classes is only unidirectional, yet two classes uses eachothers' features. Solution of this problem is (Change Unidirectional Association to Bidirectional, n.d.):

- Add the missing association to the class that needs it.

6.3.5 Change Bidirectional Association to Unidirectional

This problem is described: There is a bidirectional association between classes, yet one of the classes does not use the other's features. Solution of this problem is : (Change Bidirectional Association to Unidirectional, n.d.)

- Remove the unnecessary association.

6.4 Simplifying Conditional Expressions Methods

Some "Simplifying Conditional Expressions" methods are: Replace Conditional with Polymorphism, Consolidate Conditional Expression, Decompose Conditional, etc.

6.4.1 Replace Conditional with Polymorphism

This problem is described: conditionals that chooses different behavior depending on the object type or properties. Solution of this problem is (Replace Conditional with Polymorphism, n.d):

- create subclasses matching the branches of the conditional.
- create a shared method and move code from the corresponding branch of the conditional to it.
- replace the conditional with the relevant method call.

6.4.2 Consolidate Conditional Expression

Consolidate Conditional Expression problem is described: multiple conditionals which the returned values are the same is seen in the bad code block. Group the conditionals using the '&&' or the '||' operators for the best solution and cleaner code. After that, extract the code into a separate function. In other words, consolidate all these conditionals in a single expression (Consolidate Conditional Expression, n.d).

6.4.3 Decompose Conditional

Decompose Conditional problem is having a complex conditionals such as switch, if-then-else statements. Best solution for Decompose Conditional problem is (Decompose Conditional, n.d.) :

- Decompose the complex parts of the conditional into separate methods.

6.5 Making Method Calls Simpler Methods

Some "Making Method Calls Simpler" methods are: Rename Method, Add Parameter, Remove Parameter, Replace Parameter with Explicit Methods, Preserve Whole Object, Hide Method, etc.

6.5.1 Rename Method

Method naming is important for better coding. The name of a method must explain what the method does. Solution of this problem:

- rename the method.

Rename Method Refactoring approach improves the code readability. For example; change the findNumber() method name as a getEmployeeID().

6.5.2 Add Parameter

A method needs more information from its caller to perform certain actions. Create a new parameter to pass the information (Add Parameter, n.d.). For example; change the method getExpirationDate() method as a getExpirationDate(Date).

6.5.3 Remove Parameter

If a parameter is not used in the body of a method, remove the unnecassary parameter (Remove Parameter, n.d.). For example; change the method getExpirationDate(Date) method as a getExpirationDate().

6.5.4 Replace Parameter with Explicit Methods

This problem is described: a method is split into parts, each of which is run depending on the value of a parameter. Solution of this problem (Replace Parameter with Explicit Methods, n.d.) :

- Extract the individual parts of the method into their own methods and call them instead of the original method.

6.5.5 Preserve Whole Object

Preserve Whole Object problem can solved by passing the whole object instead of passing parameters to a method.

6.5.6 Hide Method

This problem is described: a method is not used by other classes or is used only inside its own class hierarchy. Solution of this problem (Hide Method, n.d.)

- make the method private or protected.

Hide method problem is shown in Figure 6.19.

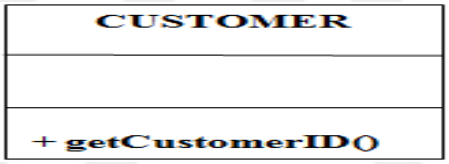

Figure 6.19 Bad design sample for hide method refactoring.

Solution of hide method problem is shown in Figure 6.20.

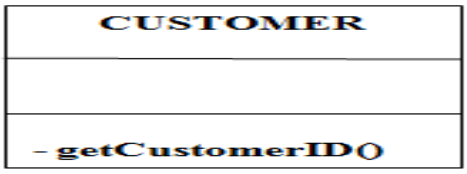

Figure 6.20 Good design sample hide method refactoring.

6.6 Dealing with Generalization Methods

Some “Dealing with Generalization” methods are: Pull Up Field, Pull Up Method, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass etc.

6.6.1 Pull Up Field

If two classes have the same field, Pull Up Field problem occurs. This problem can be solved by removing the field from subclasses and moving it to the superclass. This method destroys duplication of fields in subclasses. Bad design sample is shown in Figure 6.21.

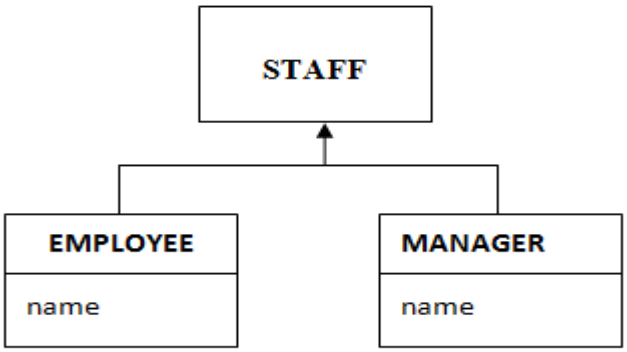

Figure 6.21 Bad design sample for pull up field.

Good design sample is shown in Figure 6.22.

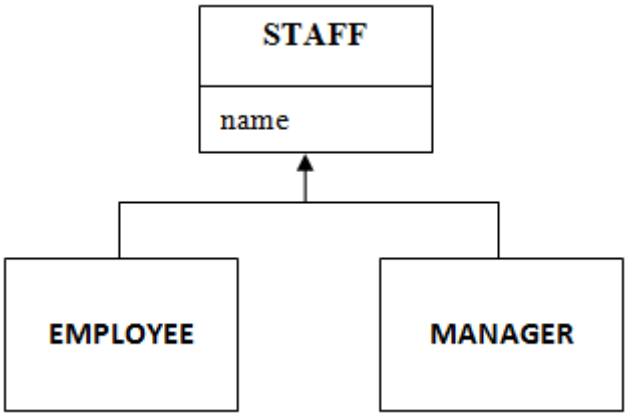

Figure 6.22 Good design sample for pull up field.

6.6.2 Pull Up Method

If subclasses which have methods that perform similar work, pull up method problem occurs. This problem can be solved by making the methods unique and then moving them to the relevant superclass. Bad design sample is shown in Figure 6.23.

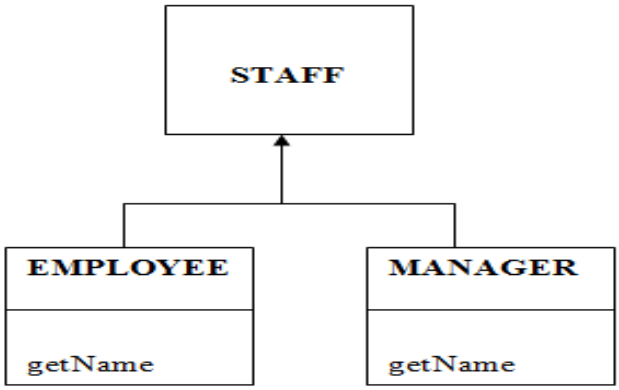

Figure 6.23 Bad design sample for pull up method.

Good design sample is shown in Figure 6.24.

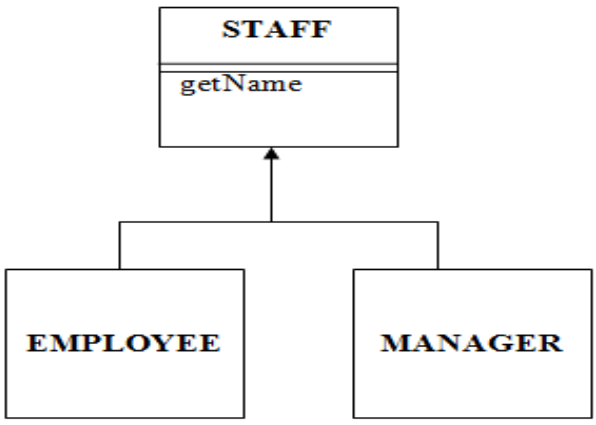

Figure 6.24 Good design sample for pull up method.

6.6.3 Push Down Method

The behavior implemented in a superclass used by only one or a few subclasses. This can be solved by moving the behavior to the subclasses. This method improves class coherence (Push Down Method, n.d.). Bad design sample is shown in Figure 6.25.

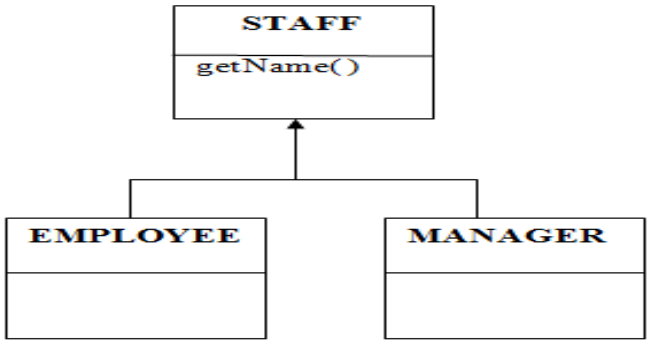

Figure 6.25 Bad design sample for push down method.

Good design sample is shown in Figure 6.26.

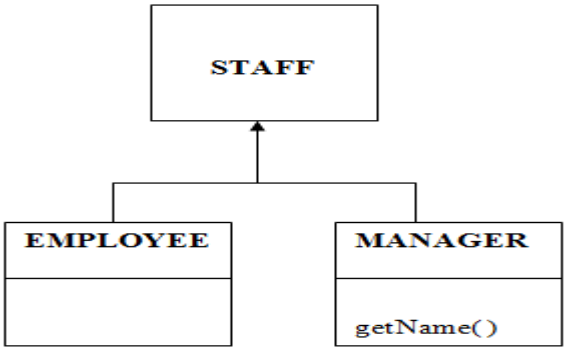

Figure 6.26 Good design sample for push down method.

6.6.4 Push Down Field

A field used only in a few subclasses so push down field problem occurs. Solution of this problem:

- Move the field to these subclasses.

Bad design sample is shown in Figure 6.27.

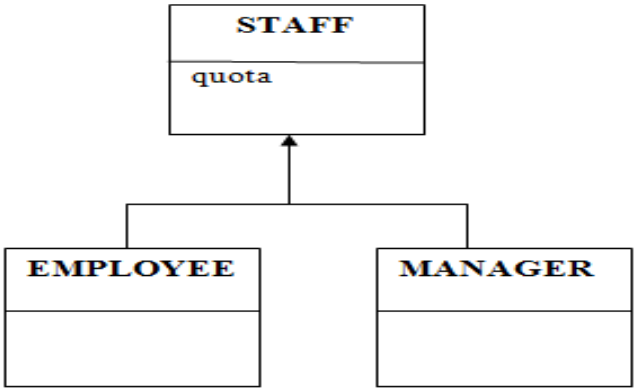

Figure 6.27 Bad design sample for push down field.

Good design sample is shown in Figure 6.28.

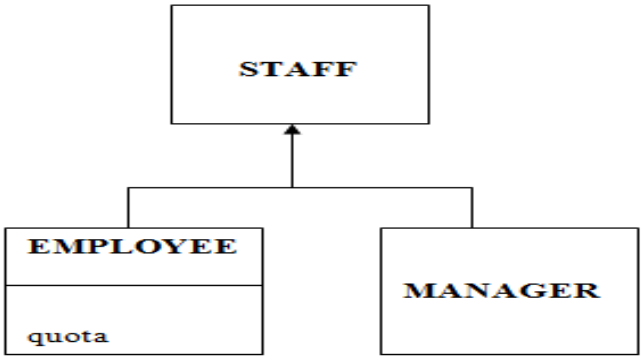

Figure 6.28 Good design sample for push down field.

6.6.5 Extract Subclass

A class has features that are used only in certain cases so extract subclass problem occurs. Solution of this problem (Extract Subclass, n.d.):

- Create a subclass and use it in these case.

Bad design sample is shown in Figure 6.29.

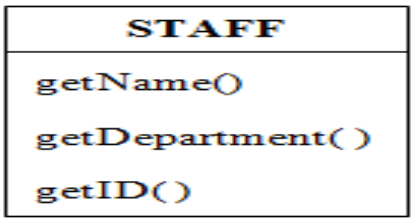

Figure 6.29 Bad design sample for extract subclass.

Good design sample is shown in Figure 6.30.

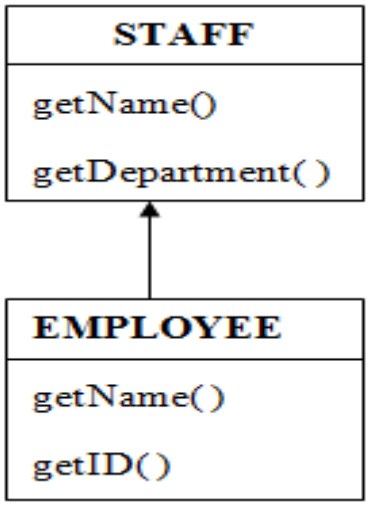

Figure 6.30 Good design sample for extract subclass.

6.6.6 Extract Superclass

Extract superclass problem means two classes with similar features. Solution of this problem is (Extract superclass, n.d.) :

- create a superclass.
- move the common features to the superclass.

Bad design sample is shown in Figure 6.31.

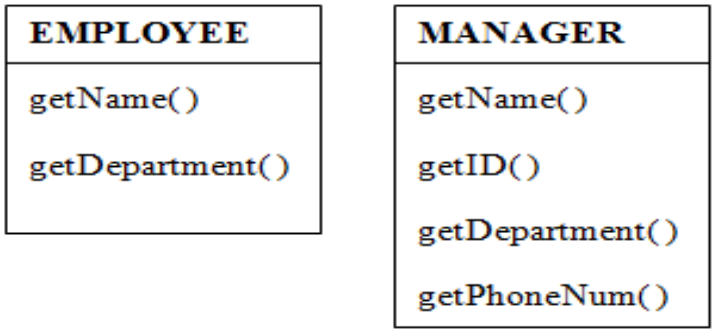

Figure 6.31 Bad design sample for extract superclass.

Good design sample is shown in Figure 6.32.

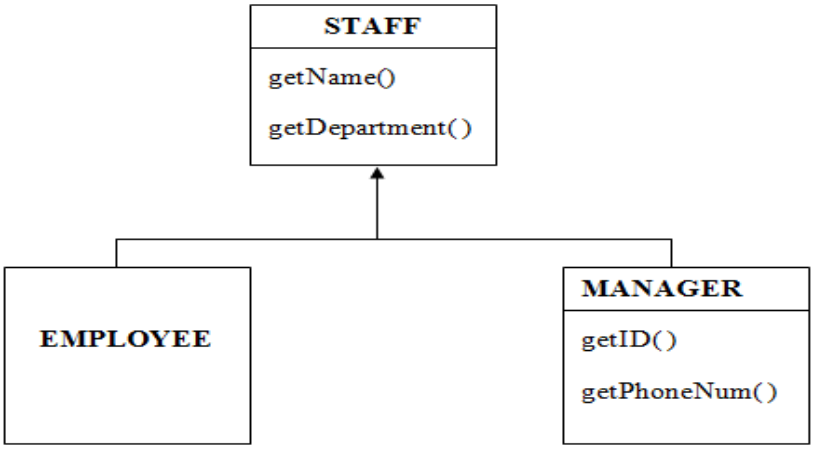

Figure 6.32 Good design sample for extract superclass.

CHAPTER SEVEN
CODE SMELLS & ALGORITHM ANALYSIS

7.1 Algorithm and Algorithm Analysis

An algorithm is a list of rules to follow in order to solve a problem and bases of computer science (Code needs algorithms, n.d.). A good algorithm aims to perform optimum performance. The good algorithm should be:

- fast,
- cover less space in memory.

Algorithm analysis is used:

- to measure algorithm's performance and timing,
- to compare different algorithms,
- to find the best available solution.

7.2 Performance of an Algorithm

Performance of an algorithm is rely on two categories which are internal and external factors. Internal factors of algorithm performance is:

- space,
- time.

External factors of algorithm performance is:

- the size of the input data,
- the computer speed,
- the quality of compiler.

Code smells reduces performance of software. Loss of performance causes somethings:

- increased costs,
- motivation disorders,
- hard transportability,

- difficult interchangeability
- poor readability of code.

In order to improve the performance of the software, Refactoring should be used.

An algorithm's performance can explain with Big-O notation mathematical approach. This mathematical approach is define the performance of an algorithm by using the internal details of an algorithm. Big-O notation shows the growth rate of an algorithm. Also, growth rate is the best indicator to reveal the performance of an algorithm.

7.3 Algorithm Analysis and Code Smells

Defects in the code can be determined by deciding through the experience of using a software tool or software engineer. Whether the code defects identify correctly or not is verified by “Algorithm Analysis” approach. Therefore, "Algorithm Analysis" approach can be used to idetify very long cycles, extreme nested loop structures, method with extreme parameter and God Class by using space, time and size of the input data factors.

This study compares some algorithm’s performance using runtime calculate approach. Also, this study investigates which method is better for certain sorting algorithms recursive or iteratives?

The first algorithm is “Bubble Sort Algorithm”. Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted (Lazy Class, n.d.). Two different solutions such as Recursive Solution and Non-Recursive Solution were examined for Bubble Sort Algorithm. Running with an application written in C # programming language to analyze Bubble Sort Algorithm solutions.

All applications within this thesis has been implemented using Windows 7 Ultimate (64 bit) operating system, Intel Core Duo 2.53 GHz CPU and 3 GB Ram.

Recursive and Non-Recursive solutions were analyzed by analysis of algorithms. Generally, Bubble Sort complexity is $O(n^2)$ and Bubble Sort works well with either linked lists or arrays $O(n^3)$. Also, Bubble sort does not work optimum with either Recursive or Non-Recursive functions. Due to dynamically changeable linked list, the performances of the software with recursive and non-recursive solution were not optimal in the current state. In this study, performance of the method calculated for the Recursive and Non-Recursive function. As a result, the Recursive solution has been determined as more optimum than Non-Recursive one, after measuring the performance of these methods. The performance of these methods were calculated as:

- Normal method: 0.0203417 seconds.
- Recursive method: 0.011363 seconds.

The second algorithm is “Quick Sort Algorithm” which is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Then, quick sort can recursively sort the sub-arrays. Two different solutions such as Recursive Solution and Iterative Solution were examined for Quick Sort Algorithm. Running with an application written in C # programming language to analyze all sorting algorithm solutions.

Quicksort is a comparison sort, so it can sort items of any type for which a "less-than" relation. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is not frequent (Quicksort, n.d.). Respectively, the result of recursive and iterative quick sort algorithm is shown as follows:

- Recursive method: 1.6E-06 seconds.
- Iterative method: 2.4E-06 seconds.

The third algorithm is “Merge Sort Algorithm”, which uses divide and conquer methods. This algorithm is an efficient, general-purpose, comparison-based sorting algorithm. The merge sort works as follows Merge Sort, n.d.):

i. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
ii. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is T(n), then, the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm. In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than $(n \lceil \log n \rceil - 2\lceil \log n \rceil + 1)$, which is between $(n \log n - n + 1)$ and $(n \log n + n + O(\log n))$.
Respectively, the result of recursive and iterative merge sort algorithm is shown as follows:

- Recursive method: 1.2E-06 seconds.
- Iterative method: 1.6E-06 seconds.

Another application program was developed for calculating timing for Insertion Sort, Selection Sort, Bubble Sort, and Quick Sort. This application allows us to test various sizes of arrays (C# Sorting Algorithms Performance Comparison, n.d.). This program uses randomly scrambled numbers. The program aims to demonstrate timing of Sorting Algorithms with an array size change option. In Table 7.1, the program tests each sorting algorithm with an 10000 array size. The program will calculate timing for each Sorting Algorithm by changing array size in each iteration (Each iteration begins with Selection Sort and ends after Insertion Sort).

Table 7.1 The array size is 10000 for each sorting algorithm.

	Array Size	**Sort Time**
Selection Sort	10000	1.154 seconds
Bubble Sort	10000	1.092 seconds
Quick Sort	10000	0.297 seconds

After that, the array size set 20000. The result for 20000 items is shown in Table 2.

Table 7.2 The array size is 20000 for each sorting algorithm.

	Array Size	**Sort Time**
Selection Sort	20000	6.584 seconds
Bubble Sort	20000	8.143 seconds
Quick Sort	20000	0.655 seconds

As a result, each Sorting Algorithm's total timing tested with different array size. The results shown in Table 7.3:

Table 7.3 Sorting algorithm's timing with different array sizes.

Array Size	**Selection Sort**	**Bubble Sort**	**Quick Sort**
10000	1.154	1.092	0.292
20000	6.584	8.143	0.655
23000	6.864	11.513	0.608
25000	6.801	7.114	0.546
5000	0.327	0.265	0.171
2600	0.109	0.093	0.047
30000	9.188	9.828	0.749
50000	25.569	43.914	1.201
67000	138.17	89.747	1.513
100000	210.477	356.384	5.304

7.3.1 Example of Algorithm Analysis with God Class

Refactoring aims to improve software performance. In this example, rectangular area calculation performed in two ways. Time complexity of bad design sample is 12. (Each line has a one complexity.) Bad design sample is shown in Figure 7.1:

```
static int areaCalculate(int edge1, int edge2)
{
     int area = edge1 * edge2;
     return area;
}
static void Main(string[] args)
{
     int shortEdge;
     int longEdge;
     int area;
     Console.Write("enter short edge: ");
     shortEdge = Convert.ToInt32(Console.ReadLine());
     Console.Write("enter long edge: ");
     longEdge = Convert.ToInt32(Console.ReadLine());
     area = areaCalculate(shortEdge, longEdge);
     Console.Write("Rectangular Area = " + area);
     Console.ReadKey();
}
```

Figure 7.1 Bad design sample for god class.

Refactoring was performed for better code readability and high performance. Class hierarchy is implemented to reach the good design. Also, good design sample is shown in Figure 7.2:

```
static void Main(string[] args)
{
     rectangular rect = new rectangular();
     rect.Height =Convert.ToInt32(Console.ReadLine());
     rect.Width= Convert.ToInt32(Console.ReadLine());
     Console.WriteLine(rect.area());
     Console.ReadKey();
}
```

Figure 7.2 Good design sample for god class.

Added class structure is shown in Figure 7.3:

```
class rectangular
{
    private int height;
    public int Height { get; set; }

    private int width;
    public int Width { get; set; }

    public int area()
    {
        int result = width * height;
        return result;
    }
}
```

Figure 7.3 Added class structure for good design.

Time complexity of good design sample is 11. (Each line has a one complexity.) Thus, we analyzed the code and calculate the time complexity of bad and good code design sample. Thus, we have found code smells by using “Algorithm Analysis” approach with time complexity. Time complexity of bad code design sample and good or refactored code design sample is different. Refactored code design sample performance is better than bad code design sample. Thus, we have proved the necessity of Refactoring. In addition, we have proved the code can be determined by algorithm analysis time complexity calculation.

CHAPTER EIGHT

CONCLUSION & FUTURE WORK

This study is prepared based on the software devolopment process and software engineering discipline. Also, in this thesis it is emphasized that code smells occurs due to omissions and negligence in system analysis, decision-making and implementation phases.

According to the Refactoring method code smells are detected with the help of a software engineer 's point of view or the software tool and bad smells can be solved manually or using the software tool. However, at this point, some problems may ocur, first, making wrong decisions by software engineers, and second, code smell detection and destruction process is too long with software engineers' experiences. One by one and step by step detection and correction of defects by using software engineers' point of view takes a lot of time. Thus, all of these causes loss of cost and labor.

In previous studies, detecting bad smells in the code by using software tools that not contain all of the available Refactoring method, which can detect only a certain part and are seen to be a solution. The most important reason of this condition is that the development rate of software tools and the rate of emergence of new code defects are not at the same or close speed.

In this study, features of previous works have examined in details; detection of bad smells in the code has been tested on some algorithms by evaulating algorithm runtimes. It was seen that code smell detection and correction defends some rules for some algorithms have better performance and thus, some of them have better and more clear code. This study compared basic sorting algorithms' performance using runtime calculation approach. Also, it has been investigated that which method is better for certain sorting algorithms recursive or iteratives in terms of their own performance with the criteria of calculated runtime by using special feature. In addition, timing of Sorting Algorithms tested with different array size and compared.

It is shown that, "Algorithm Analysis" approach can be used to idetify very long cycles, extreme nested loop structures, method with extreme parameter and God Class by using internal and external factors of algorithm performances.

In future, implementation of a system for detecting bad smells in codes has been thought by using “Algorithm Analysis” approach.

REFERENCES

Add parameter. (n.d.). Retrieved June 17, 2016 from http://refactoring.com/catalog/addParameter.html

Booch, G., (2014). To code or not to code, that is the question. *IEEE Software*, 9-11. Retrieved October 19, 2015.

Buschmann, F., (2011). Gardening your architecture, Part 1:Refactoring. *IEEE Transactions on Software Engineering*, 92-94.

Change Bidirectional Association to Unidirectional. (n.d.). Retrieved June 16, 2016 from https://sourcemaking.com/refactoring/change-bidirectional-association-to-unidirectional

Change Unidirectional Association to Bidirectional. (n.d.). Retrieved June 16, 2016 from https://sourcemaking.com/refactoring/change-unidirectional-association-to-bidirectional

Chatzigeorgiou, A., & Manakos, A., (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations System Software Engineering, 10,* 3-18.

Code needs algorithms. (n.d.). Retrieved July 18, 2016 from http://www.bbc.co.uk/guides/z3whpv4#zx3dwmn

Composing methods. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/composing-methods

Consolidate conditional expression. (n.d.). Retrieved June 16, 2016 from https://sourcemaking.com/refactoring/consolidate-conditional-expression

C# sorting algorithms performance comparison. (n.d.). Retrieved January 8, 2016 from https://www.exchangecore.com/blog/c-sharp-sorting-algorithms-performance-comparison-selection-sort-vs-insertion-sort-vs-bubble-sort-vs-quick-sort/

Decompose conditional. (n.d.). Retrieved June 16, 2016 from https://sourcemaking.com/refactoring/decompose-conditional

Devarakonda, M., (1998). The practical aspects of object-oriented programming. *IEEE Concurrency*, 6, 30-33.

Dyke, T., & Kunz, J., (1989). Object-oriented programming. *IBM Systems Journal, 28,* 465-478.

Extract subclass. (n.d.). Retrieved June 17, 2016 from https://sourcemaking.com/refactoring/extract-subclass

Extract superclass. (n.d.). Retrieved June 17, 2016 from http://refactoring.com/catalog/extractSuperclass.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., (2002). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Garrido, A. & Rosst, G., (2011). Refactoring for usability in web applications. *IEEE Software,* 60-67.

Güzel, A. & Aktaş, Ö. (2016). A survey on bad smells in codes and usage of algorithm analysis. *International Journal of Computer Science and Software Engineering, 5*, 114-118.

Güzel, A. & Aktaş, Ö. (2015). Kodlardaki kötü kokuları tespit etme yöntemleri ve algoritma analizi. *Akademik Bilişim 2015* (In Press).

Hide method. (n.d.). Retrieved June 17, 2016 from https://sourcemaking.com/refactoring/hide-method

Inline method. (n.d.). Retrieved June 15, 2016 from http://www.skorkin.com/2011/02/refactorings-inline-method/#.V3KO4PmLTIU

Inline temp. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/inline-temp

Kessentini, M., Mahaouachi, R., & Ghedira, K., (2013). What you like in design use to correct bad-smells. *Software Qual Journal, 21,* 551-571.

Khomh, F., Vaucher, S., Gueheneuc, Y., & Sahraoui, H., (2009). A bayesian approach for the detection of code and design smells. *Ninth International Conference on Quality Software,* 305-314.

Kim, M., & Notkin, D., (2005). Using a clone genealogy Extractor for understanding and supporting evolution of code clones. *ACM SIGSOFT,* 30, 1-5.

Liu, H., Zhiyi, M., Shao, W., & Niu, Z., (2012). Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering, 38,* 220-235.

Lazy class. (n.d.). Retrieved January 3, 2016 from https://sourcemaking.com/refactoring/smells/lazy-class

Liu, H., Liu, Y., Xue, G., & Gao, Y., (2014). Case study on software refactoring tactics. *IET Software, 8,* 1-11.

Li, W., & Shatnawi, R., (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *The Journal of Systems and Software, 80,* 1120–1128.

Malhotra, R., & Pritam, N., (2012). Assessment of code smells for predicting c change proneness. *Software Quality Professional, 15,* 33-40.

Mantyla, M., & Lassenius, C., (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Empir Software Engineering, 11,* 395-431.

Mens, T., & Tourwe´, T., (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30,* 126-139.

Merge sort. (n.d.). Retrieved January 5, 2016 from https://en.wikipedia.org/wiki/Merge_sort

Moha, M. (2007). Detection and correction of design defects in object-oriented designs. *OOPSLA'07, Canada,* 949-950.

Move method. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/move-method

Move field. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/move-field

Moving features between objects. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/moving-features-between-objects

Murphy-Hill, E., Parnin, C. & Black, A., (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering, 38,* 5-18.

Push down method. (n.d.). Retrieved June 17, 2016 from https://sourcemaking.com/refactoring/push-down-method

Quicksort. (n.d.). Retrieved January 4, 2016 from https://en.wikipedia.org/wiki/Quicksort

Rahman, F., Bird, C. & Devanbu, P., (2012). Clones: what is that smell?. *Empir Software Eng, 17,* 503–530.

Rech, J. & Schäfer, W., (2007). Visual support of software engineers during development and maintenance. *ACM SIGSOFT Software Engineering Notes, 32. 1-3.*

Refactoring 1: consolidating conditional expressions. (n.d.). Retrieved June 16, 2016 from http://www.codediesel.com/software/refactoring-1-consolidate-conditional-expression/

Remove parameter. (n.d.). Retrieved June 17, 2016 from https://sourcemaking.com/refactoring/remove-parameter

Replace conditional with polymorphism. (n.d.). Retrieved June 16, 2016 from https://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

Replace parameter with explicit methods. (n.d.). Retrieved June 17, 2016 from https://sourcemaking.com/refactoring/replace-parameter-with-explicit-methods

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M., (2010). Building empirical support for automated code smell detection. *Empirical Software Engineering and Measurement 2010, Italy*.

Sjøberg, D., Yamashita, A., Anda, B., Mockus, A. & Dyba, T., (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering, 39,* 1144-1156.

Split temporary variable. (n.d.). Retrieved June 15, 2016 from http://debuggable.com/posts/composing-methods-split-temporary-variable:480f4dfe-1e28-4e32-b4b3-458ccbdd56cb

Split temporary variable. (n.d.). Retrieved June 15, 2016 from https://sourcemaking.com/refactoring/split-temporary-variable

Stroustrup, B., (1998). What is object-oriented programming?. *IEEE Software,* 10-20.

Thompson, S., & Li, H., (2013). Refactoring tools for functional languages. *Cambridge University Press*, 23, 293-350.

Wirfs-Brock, R. J., (2008). Connecting design with code. *IEEE Software,* 20-21.

Yamashita, A., (2014). Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empir Software Engineering, 19,* 1111–1143.