An Enhancement Approach of Software System

-Using Reverse Engineering and Restructuring Concept to Improve the Quality of Software Code

By
Hamza Ali Abdelrahman El-Ghadhafi

Supervisor
Dr.Tawfig Eltawil

This Thesis was submitted in Partial Fulfillment of the Requirements for Master's Degree of Computer.

University of Benghazi

Faculty of Information Technology

April 2018

Copyright © 2018.All rights reserved, no part of this thesis may be reproduced in any form, electronic or mechanical, including photocopy , recording scanning , or any information , without the permission in writhing from the author or the Directorate of Graduate Studies and Training university of Benghazi.

حقوق الطبع 2018 محفوظة . لا يسمح اخذ اى معلومة من اى جزء من هذه الرسالة على هيئة نسخة الكترونية او ميكانيكية بطريقة التصوير او التسجيل او المسح من دون الحصول على إذن كتابي من المؤلف أو إدارة الدارسات العليا و التدريب جامعة بنغازي.

University of Benghazi

Faculty of information Technology

Department of Computer science

An Enhancement Approach for the Quality of Software Cod.

- Using Reverse Engineering and Restructuring Concept to Improve the Quality of Software Code

By

Hamza Ali Abdelrahman El-Ghadhafi

This Thesis was Successfully Defended and Approved on **26.4.2018**

Supervisor

Dr. Tawfig M. Eltaweel

Signature: ....

Dr. Omar Mustafa Elsalabi (**Internal examiner**)

Signature: ....

Dr. Mohamed Ahmed Khlaif (**External examiner**)

Signature: ....

(Dean of Faculty) **(Director of Graduate studies and training)**

Acknowledgements

" Great things are not done by impulse, but by series of small things brought together..."

— Vincent van Gogh

It gives me a great deal of pleasure to express my profound gratitude to my thesis advisor Dr. Tawfig Tawill for his persistent and inspiring supervision and for his valuable advice and guidance in shaping my research towards a this successful thesis isa..

I am grateful to the members of my family, relatives and friends, especially to my mother Fitma Salam, , and my sisters Fathia and Ebtesam Elgathafi, who did not enjoy the share of my time and attention that they deserved. For those that I have not explicitly mentioned here, thank you for being part of this thesis and helping me grow as a person and a researcher. Above all, my sincere gratitude to the Almighty, who creates and makes things happen.

Table of Contents

Copyright © 2018......ii

Examination Commitee......iii

Acknowledgements......iv

List of Tables......x

List of Figures......xi

List of Appendices......xiv

Abstrct......xv

Chapter 1 – Introduction

Introduction to SDLC......1

Problem Statement......3

Motivation......4

Research Amis......5

Research questions......6

Scope and Limitation of the Thesis......6

Significance of the Thesis......6

Organization of the chapters of Thesis......7

Chapter 2 – Background

Introduction......8

Code Smells 8

Smells Definition......9

Classification of the Code Smells......11

The Problem of Duplicated Code......13

The Origin of Duplicated Code......13

Classification of Duplicated Code......15

The Problem of Long Method 16
The Problem of Large Class....17
Side Effects Of Code Smells....17
Software Performance....17
Detection of Bad Smells 18
The Problem....19
Previous Work....19

Chapter 3 – An Enhancement Approach of Software System

Introduction....21
Smells Detection Technique or Process....21
Constraints....21
Proposed Approach....24
Exploration and Assessement Stage....24
Preprocessing The Code....25
Suspend Code Conventions....25
Methods of Suspend Code Conventions....26
Code Filtering....27
Code Formatting....29
Mapping Code with UML Diagrams....30
Reverse Engineering of Software....30
Reasons for Reverse Engineering....31
Reverse Engineering Types....32
UML Formatting....32
Code Restructuring Stage....35
Detection of Code Smells....36
The Potential Cases of Code Clone....37

Duplication in the Same Method..37

Duplication in the Same Class..38

Duplication between Sibling Classes..39

Duplication with Super class...40

Duplication with Ancestor...41

Duplication with First Cousin...42

Duplication in Unrelated Classes..43

The Potential Cases of Long Method...43

Long Method with More than 10 Lines...44

Long Method because Duplicate Lines...45

Long Method with Loops...46

Long Method with Conditional Expressions..47

The Potential Cases of Large Class...47

Large Class with Many Methods..48

Enhancement Mechanism...49

Restructuring Template...49

Application Solution...49

Summary...51

Chapter 4 – Case Study

Introduction...53

Exploration and Assessement Stage...53

Suspend Code Conventions..54

Code Filtering..54

Code Formatting..57

UML Formatting..63

Code Restructuring Stage..65

Detection of Code Smells...65

The Potential Cases of Code Smell..65

At the beginning, the simple look of the system (The bird's look)................66

At the ending, the close look of the system (The infrastructure inspect)........70

Enhancement Mechanism...72

Rename Method...72

Extract Method...74

Extract Method...75

Application Solution...78

Chapter 5 – The Quantitative Validation of the Enhancement Approach

Introduction...79

Presentation of the Results...79

Project Analyzer Tool...80

Project Metrics...80

Project Status Report...81

System Size...81

Commentation...82

Complexity...82

Conditional Nesting...83

Procedure Length...83

File Lengt...84

Parameters...85

Class Design...85

Coupling Metrics...86

Chapter 6 – Conclusions and Future work

Analysis...87

Conclusion..89

Future Work..90

Bibliography...91

Appendices...98

List of Tables

Table 1 classfication of code smells..11

Table 2 befor remove the blanks from the source code..28

Table 3 after remove the blanks from the source code...28

Table 4 the determining the restructuring units of the system..................................30

Table 5 the summary of code duplication restructuring mechanisms........................51

Table 6 the summary of long method and large class restructuring mechanisms...52

Table 7 befor remove the blanks from the source code..55

Table 8 after remove the blanks from the source code...56

Table 9 the determining the restructuring unit for the Main form............................59

Table 10 the determining the restructuring unit for the Card_items form.................60

Table 11 the determining the restructuring unit for the Edn_Etlaf_bill form............60

Table 12 the determining the restructuring unit for the Edn_Srt_bill form...............60

Table 13 the determining the restructuring unit for the Loing_form form................61

Table 14 the determining the restructuring unit for the store form...........................61

Table 15 the determining the restructuring unit for the suppliers form.....................62

Table 16 the determining the restructuring unit for the users form...........................62

List of Figures

Figure 3.1: the Abstract Diagram of our Approach..22

Figure 3.2: the Schematic Diagram of Enhancement Approach...............................23

Figure 3.3: the process of remove (removal) uninteresting parts..............................26

Figure 3.4: the UML class diagram that represent a class/interface/form..................32

Figure 3.5: the generalization relationship between two classes/interfaces/forms.....33

Figure 3.6: the interface realization (implement) relationship between a class and an interface...33

Figure 3.7: the directed association relationship between two classes.......................33

Figure 3.8: the instantiate dependency relationship between two classes..................34

Figure 3.9: the usage dependency relationship between two classes.........................34

Figure 3.10: the methods in the class or interface...34

Figure 3.11: the example for the process of UML Formatting (transformation to an appropriate intermediate representation) used in the Enhancement Approach...35

Figure 3.12: the duplication in the same method..37

Figure 3.13: the duplication in the same class...38

Figure 3.14: the duplication between sibling classes..39

Figure 3.15: the duplication with superclass...40

Figure 3.16: the duplication with ancestor..41

Figure 3.17: the duplication with first cousin...42

Figure 3.18: the duplication in Unrelated Classes...43

Figure 3.19: the long method with more than 10 lines..44

Figure 3.20: the long method because duplicate lines..45

Figure 3.21: the long method with loops..46

Figure 3.22: the long method with conditional expressions......................................47

Figure 3.23: the large class with many methods..48

Figure 4.24: the code conventions are suspended...54

Figure 4.25: the appropriate intermediate representation of the case study using the Project Analyst application...63

Figure 4.26: the detail description of appropriate intermediate representation of the case study using...64

Figure 4.27: the similarity of the methods' names are in the forms..........................66

Figure 4.28: the method that is considered as a long methods.................................67

Figure 4.29: the class is considered a large class...69

Figure 4.30: the similarity of the code lines that are in the different methods but in the same form..70

Figure 4.31: the similarity of the some methods that are found in the different forms...71

Figure 4.32: the apply of Rename Method mechanism...73

Figure 4.33: the Extract Method mechanism: Create a new method in the same class [i.e. clear()]..74

Figure 4.34: the Extract Method mechanism: Copy the extracted code from the source method into the new method..74

Figure 4.35: the apply of Extract Method mechanism...75

Figure 4.36: the Extract Method mechanism: Create a new method in the same class [i.e. quntity1()]..76

Figure 4.37: the Extract Method mechanism: Copy the extracted code from the source method into the new method..76

Figure 4.38: the Extract Method mechanism: send local variable as parameters to the new method...76

Figure 4.39: the Extract Method mechanism: define the new integer variable for return back Holds the result of this method..77

Figure 4.40: the apply of Extract Method mechanism...77

Figure 4.41: return the code that is used to link of the database................................78

Figure 4.42: return some important developer comments that have been omitte......78

Figure 5.43: Charts are illustrating the distribution of the system size before and after the implementation of the Enhancement Approach..81

Figure 5.44: the average depth of conditional nesting (DCOND).............................83

Figure 5.45: the average procedure length (LINES/proc)...84

Figure 5.46: the average file length(LINES/file) 84

Figure 5.47: the average number of procedure parameters (PARAMS)...................85

Figure 5.48: the coupling metrics...86

List of Appendices

Appendix A: Restructuring Process 98

An Enhancement Approach of Software System

-Using Reverse Engineering and Restructuring Concept to Improve the Quality of Software Code

By

Hamza Ali Abdelrahman El-Ghadhafi

Supervisor

Dr. Tawfig M Eltawil

Abstract

Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development for improved productivity. As a result, software systems often have similar segments of code, called software clones or code clones. Due to many reasons, unintentional smells may also appear in the source code without awareness of the developer. Studies indicate that the term of code smell leads to indicate violation of fundamental design principles for software code and negatively impact design quality; consequently, this code becomes very difficult for developers to comprehend. This of course makes developers spend much more time to boost the code; and the maintenance process becomes very expensive. This thesis describes an approach which allows to detect the code smells from source code and removal of these smells for refinement and improvment the quality of software system taking into account keeping the external behaviour of software system, and judging the efficiency of systems code. Consequently, we develop an approach which allows the enhancement of software systems from a source code. The Enhancement Approach is based on the concept of reverse engineering ,which is used to describe the software code by UML diagrams, In order to facilitate the process to identify the situations of each code smell.

In addition, based on the concept of restructuring the process of changing a software system in such a way that does not alter the external behaviour of the code is yet improves its internal structure. Finally, The concept of situation is a set of applicable restructuring, which is associated with a given situation.

The principle of our approach is to find for each smell situation and to propose a list of possible restructuring.

CHAPTER 1
Introduction

1.1 Introduction

In software engineering, a software development is a splitting of software development work into distinct phases (or stages) containing activities with the intent of better planning and management [80]. It is often considered as a process used by software industry to develop high quality softwares. The Software Development aims to produce a high quality software that meets customer expectations, reaches completion within times and cost estimates [22].

Software development is a fundamental process of program designing and other related processes such as code programming, documenting, testing, and bug fixing. It is also involved in creating and maintaining applications among many software products. It is also known as a system development methodology, software development process, software process, software development model, software development life cycle (SDLC). In fact, a wide variety of processes have been exclusively developed over the last decade. Each has its own recognized strengths and weaknesses [21] [22]. One software development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations [24][35].

Most methodologies have so much in common, including the following essential phases of software development [18]:

- ✓ Requirements Engineering phase.
- ✓ Design phase.
- ✓ Implementation or Coding phase.
- ✓ Testing phase.
- ✓ Deployment phase.
- ✓ Maintenance and Bug Fixing phase.

The software development methodology or life cycle(SDLC) is a framework defining tasks performed at each step of the software industry. There may be many additional steps and stages depending upon the nature of the software product. You may have to go through multiple cycles during the testing phase as software testers find problems and bugs, and developers fix them before a software product is officially released [30].

A software development process makes everything easier and reduces the amount of problems encountered [79]. Each phase produces feedback that affects the next phase. For instance, the requirements gathered during the requirements phase influence the design, which is translated into working software code during the implementation phase. The software code is verified against the requirements during the testing phase. Then the complete software product is delivered to customer in deployment phase. The actual problems or bugs that come up when the customers starts using the software system are solved during maintenance phase [23][39].

The focus of this research on some of the existing problems in both the implementation and testing phase that have negative impact on the software code design lead to production of poor-quality software negatively impact design quality.

In fact, the implementation phase has one key activity: ***Writing code*** for a program. This is considered as one of critical factors in creating truly successful software development." A good or poor design of software relies heavily on a quality of the code design " [39]. Moreover, writing code is widely considered to be one of the longest tasks in software engineering process.

There are two different techniques to *writing clean code*, regardless of what programming language you are working on, that are: ***Use Your Brain*** and ***Copy And Paste Method*** [78].

The first technique (***Use Your Brain***): Instead of simply copying and pasting code from Google or any other source, learn to use your brain for writing your code. Use the help(in the programming language) that you are getting to your advantage and try to optimize the code that you have. Simply using others code might give you a

temporary joy, but you will not have the satisfaction until you're able to write code and solve problems by yourself. The second technique for writing code that is widely used techniques in this phase known as : ***Copy and Paste method***. The majority of developers have been utilizing such a popular method for writing code due to a number of main advantages , most importantly being simple to use. Another major advantage is the fact that such a method is often used for less time-consuming and cost when developing the software system. Therefore, a final software product is delivered in the shortest possible time with relatively minimum cost. However, using this technique can cause many problems which frequently appear in the source code. These problems are known as *Code Smells*.

1.2 Problem Statement:

The term "*code smell*" was introduced by ***Kent Beck*** to define those structural problems in the source code that can be detected by experienced developers. According to Martin Fowler, " a code smell is a surface indication that usually corresponds to a deeper problem in the system "[17]; Girish Suryanarayana and et al define a smell / smells as " certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality " [27]. Smells do not prevent the current program from functioning. Instead," they indicate weaknesses in design that may consume computer resources, i.e. execution time and memory, increasing the risk of bugs or failures in the future" [17]. When one of the smell problems mentioned above exists in the software code, there will be need for maintenance in order to develop the code.

Due to changing of system requirements and growing need for software improvement, modifying legacy systems have become more complex and expensive tasks, because of time-consuming process of program comprehension. Thus, there is a need for software engineering methods and tools that facilitate program understanding[7].

Generally, the need for maintaining existing software systems has become an important business goal in recent years in order to develop software efficiency, performance, maintainability, reusability and scalability [2].

1.3 Motivation

Over the last decade , the software engineering community had to encounter a number of rising issues and common problems related to performing system function such as understanding legacy code and consuming computer resources. In fact, software engineering has undergone a paradigm shift as the size of the software systems increased dramatically and businesses began to rely increasingly on computers and information systems. Therefore, a substantial portion of the software development effort is spent on maintaining and improving existing systems rather than developing new ones.

Having stated the fact that there has been a clear growing increase in the size of information systems ,here means there is a parallel increase in the size of system code, too. Unfortunately, this definitely makes the system code becomes more complex [16]. This complexity allows unwelcome code smells to be present in the source code, and eventually has a negative effect on the quality of system. Actually, this argument was supported by Prajakta Ashtaputre and et al, who pointed out that "the presence of these code smells may weaken the quality of design structure as well as software quality such as changeability, maintainability, understandability and readability" [8]. Moreover, the presence of code smells can warn about wider development problems such as wrong architectural choices or even bad management practices. The result of that, day by day the complexity levels of Software system increasing [82]. Hence, more effort is required for software organizations to develop new or rebuild existing system of high quality.

Furthermore, If code is poorly designed due to this composite system, then this code becomes very difficult for developers to comprehend. This of course makes developers spend much more time to boost the code; and the maintenance process become very expensive. This fact was well observed by Anshu Rani and Harpreet Kaur. They both pointed out that "Poorly designed software systems are difficult to understand and maintain. Software maintenance can take up to 50% of the overall development costs of producing software. One of the main attributes to these high costs is poorly designed code, which makes it difficult for developers to understand the system even before considering implementing new code" [13]. Moreover, presence

of bad smells in object-oriented software hints at its low maintainability, which can be measured with the use of various maintainability quantification metrics. Some of these metrics concern such aspects of software maintainability like coupling, cohesion, size and complexity, or description. Therefore, at least theoretically, the enhancement of the software maintainability can be identified with the reduction of bad smells [84].

Software maintenance projects are very costly. The total maintenance costs of a software project are estimated to 40%-70% of the total cost of the development lifecycle of the project. Consequently, reducing the effort spent on maintenance can be seen as a natural way of reducing the overall costs of a software project. This is one of the main reasons for the recent interest in concept such as code smells and solution of that. Doing this will increase the understandability of code, make it easier to implement new features and debug the code [2]. By providing an appropriate and effective approach which can overcome the problems in code. This premise focuses on “effectively spending time and money in order to save time and money in the future” [77]. As a consequence, it becomes very important to implement approach for detecting and removal these smells in order to refine and improve the quality of software system, taking into account the keeping of the external behaviour of software system, i.e, the functions that performed by the system.

1.4 Research Aims

The main Aims of this research are as follows:

1. introduce an overview of code problems and describe their side effects in a software program.
2. reduce the possibility of code problems by developing an effective approach to increase the quality of software.
3. Attempt to understand the approach through relevant examples and describe the proposed solutions for the existence of bad smells in software code.

These goals are reflected in this attempt to answer the following research questions*:*

1.5 Research Questions

The researcher identified four research questions that rendered relevant for this research:

RQ 1: Size- Does the existence of the code smells make the source code large? And, Does the restructuring of the source code make it smaller?

RQ 2: Complexity - Is the complexity of the system affected by the size of the smells that exist in the source code?

RQ 3: Software Reliability-Does the program work without failure after applying the suggested restructurings on the program? (Probability of failure-free operation of a computer program for a specified time in a specified environment).

RQ 4: Maintainability - How good are code smells as indicators of system-level Maintainability of software?

1.6 Scope and Limitation of the Thesis

There are different and complex challenges existing in the software code which can impair the quality of software. Thus, this research will focus on three major problems which are the main limitation of the current study. (Duplicated Code, Large class and Long Method). In addition, the Object -Oriented Programs are only covered by the proposal approach and here are considered as the secondary scope of this study.

1.7 Significance of the Thesis

- The researcher present an approach characterized by its simplicity for analyzing, detecting and restructuring duplicated code, Large class ,Long Method in an object oriented context , which is the main contribution of this thesis.

- The researcher use the relationships between the software classes constrained by the object oriented (OO) context to define bad smells depending on the basics of the UML notion :
    1. The researcher proposes a set of applicable situations for each smell to facilitate the detection process of the bad smells for developers.
    2. Each of them determines a set of applicable restructuring.
- The researcher present a solutions to the problems mentioned above are described by using textual methods.

1.8 Organization of the Chapters of Thesis

The chapters of this thesis are organized as follows:

- ✓ **Chapter one** gives a short overview about software development life cycle (SDLC), introduction of the bad code smells and it also includes the problem statement, motivation, research aims, objectives, research questions, scope and limitation and the significance of the theses.
- ✓ **Chapter two** presents a general overview about definition, types, and classification of the bad code smells and it also presents in detail the problems of duplications, Large class and Long Method including reasons of occurrence and it also includes the side effects of this problems. At the end of this chapter, the literature that is related to the proposed approach is reviewed.
- ✓ **Chapter three** presents the proposed approach for solving the problem. It also explains in detail all the steps to be taken when using UML diagrams to detect smells and restructuring techniques to solve each of these smells using textual methods.
- ✓ **Chapter four** presents the General Mills Company system as a case study to show how the proposed methodology can be used to enhance a complete system by Enhancement Approach.
- ✓ **Chapter five** presents the quantitative evaluation of the previous case study by the object-oriented metrics, using reliable tools.
- ✓ **Chapter six** presents conclusion of the research, analysis (answers of research questions)and future work.

CHAPTER 2
Background

2.1 Introduction

This chapter presents a general overview about definition, types, and classification of the bad code smells. Moreover, it presents details of three major problems (Duplicated Code, Large class and Long Method) including reasons of occurrence and it also includes the side effects of this problems. At the end of this chapter, shortcomings of some existent approaches are discussed.

2.2 Code Smells

Code smells (also known as a bad smells) are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain. Code smell in computer programming code, refers to any symptom in the source code of a program that possibly indicates problems. As written by Kent Beck:

"*A code smell is a hint that something has gone wrong somewhere in your code*"

According to Martin Fowler " *a code smell is a surface indication that usually corresponds to a deeper problem in the system* ". Another way to look at smells is with respect to principles and quality, Girish Suryanarayana and et al define a smell / smells as:

"*certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality* "[27].

Code smells are usually not bugs and Smells do not prevent the current program from functioning. Instead, "they indicate weaknesses in design that may consume computer resources, i.e. execution time and memory, increasing the risk of bugs or failures in the future" [17]. In other words, the suspect structure may not be causing serious harm (in terms of bugs and failures) at the moment, but it has a negative

impact on the overall structure of the system and as a consequence, on its quality factors. Code smells indicate that the maintainability of the specific code might not be as good as its potential, or to put it in the words of Fowler, "*Any programmer can write code that a computer can understand. Good programmers write code that humans can understand* ". lately the importance of writing understandable code, has got more focus and acceptance. Code smells can clutter the design of a system, making it harder to understand and maintain. Moreover, the presence of code smells can warn about wider development problems such as wrong architectural choices or even bad management practices. Therefore, when one of the smells problems mentioned in the next exists in the software code, there will be need for maintenance in order to develop the code in a good quality.

2.3 Smells Definition

Martin Fowler and Beck identifies 22 code smells are[17][23][26][27]:

1. **Duplicated Code:** identical or very similar code exists in more than one location.
2. **Long Method :** a method, function, or procedure that has grown too large.
3. **Large Class** :class that has grown too large. See God object.
4. **Long Parameter List:** Long parameter lists are hard to understand. You do not need to pass in everything a method needs, just enough so it can find all it needs.
5. **Divergent Change:** Software should be sutured for ease of change. If one class is changed in different ways for different reasons, it may be worth splitting the class in two so each one relates to particular kind of change.
6. **Shotgun Surgery:** If a type of program change requires lots of little code change in various different classes, it may be hard to find all the right places that do need changing.
7. **Feature Envy :**a class that uses methods of another class excessively.
8. **Data Clumps:** Sometimes you see the same bunch of data items together in various places: fields in a couple of classes, parameters to methods, local data. May be they should be grouped together into a little class.

9. **Primitive Obsession:** Sometimes it is worth turning a primitive data type in to a lightweight class to make it clear what it is for and what sort of operations are allowed on it.
10. **Switch Statements:** Switch statements tend to cause duplication. You often find similar switch statements scattered through the program in several places.
11. **Parallel Inheritance Hierarchies**: In this case, whenever you make a subclass of one class, you have to make a subclass of another one to match.
12. **Lazy Class :** a class that does too little.
13. **Speculative Generality:** Often methods or classes are designed to do things that in fact are not required.
14. **Temporary Field:** It can be confusing when some of the member variables in a class are only used occasionally.
15. **Message Chains:** A client asks one object for another object, which is then asked for another object, which is then asked for another, etc. This ties the code to a particular class structure.
16. **Middle Man:** Delegation is often useful, but sometimes it can go too far. If a class is acting as a delegate, but is performing no useful extra work, it may be possible to remove it from the hierarchy.
17. **Inappropriate Intimacy:** This is where classes seem to spend too much time delving into each other's private parts. Time to throw a bucket of cold water over them!
18. **Alternative Classes with Different Interfaces:** Classes that do similar things, but have different names, should be modified to share a common protocol.
19. **Incomplete Library Class:** It's bad form to modify the code in a library, but sometimes they don't do all they should do.
20. **Data Class:** Classes that just have data fields, and access methods, but no real behavior. If the data is public, make it private!
21. **Refused Bequest :**a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. See Liskov substitution principle.
22. **Comments:** If the comments are present in the code because the code is bad, improve the code.

2.4 Classification of the Code Smells

This section briefly introduces the higher level taxonomy for classifying the bad code smells identified by Fowler and Beck. Although, Fowler and Beck present the 22 bad smells in a single flat list and do not provide any classification of the smells.

This taxonomy makes the smells more understandable and recognizes the relationships between the smells. The classes are: bloaters, object-orientation abusers, change preventers, dispensable, and couplers [32][81]. The table below shows a description of the classes of code smells:

Group Name	Smells in Group	Description
The Bloaters	• Long Method • Large Class • Primitive Obsession • Long Parameter List • Data Clumps	Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).
The Object-Orientation Abusers	• Switch Statements • Temporary Field • Refused Bequest • Altemative Class with Different Interfaces	All these smells are incomplete or incorrect application of object-oriented programming principles.
The Change Preventers	• Divergent Change • Shotgun Surgery • Parallel Inheritance Hierarchies	These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result
The Dispensable	• Lazy class • Data class • Duplicate Code • Dead Code • Speculative Generality	A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.
The Couplers	• Feature Envy • Inappropriate Intimacy • Message Chains • Middle Man	All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

table 1 Classifications of Code Smells

In the modern methodology, software maintenance encompasses activities and processes involving existing software, not only after its delivery but also during its development. Worth mentioning is the fact that nowadays more than 80% of total software life-cycle costs is devoted to its maintenance [71].

Swanson [1976] distinguishes and describes three kinds of software maintenance:

1) **Corrective maintenance** – performed in response to processing, performance and implementation failures;
2) **Adaptive maintenance** – performed in response to changes in data and processing environments;
3) **Perfective maintenance** – performed to eliminate processing inefficiencies, enhance performance, or improve maintainability.

Although for small systems, maintenance and evolution may not be an issue; for large software systems, their effects cannot be ignored. It has been found that almost 40-80% (average 60%) of the costs of developing a typical software system is consumed on the maintenance phase, which indicates there is a need for state-of-the-art techniques, methods, and tools to support maintenance and evolution. Programmers often use code fragments by simple copy and paste them with or without adaptation. These identical code fragments are called as software clones. Due to the copy-paste habits of programmers, clones are inevitable in software development. Previous studies have reported that the total quantity of cloning in software systems varies from 5-15% and can be even 50% of the main code.

Although some positive impacts of clones have been identified, their negative impacts cannot be ignored (e.g. increased program size, update anomalies). A code fragment having a bug causes the same problem to all other fragments copied from it. Fixing the bug requires the developer to check and update all copied locations as necessary. Enhancing a code fragment also requires the developer to look for its duplicated code fragments to ensure that changes are propagated to all desired locations, which also multiplies the work need to be done. So, clones are treated as a "bad smell" in code and are a major contributor to project maintenance difficulties [71].

2.4.1 The Problem of Duplicated Code

Duplicated code (also known as code clones) is one of the malicious "code smells" that often need to be removed for enhancing maintainability. Code duplication is widely considered to be one of the factors that severely complicates the maintenance and evolution of large software systems. From the maintenance perspective, the existence of code clones may increase maintenance effort.

"*Duplicate code is a sequence of source code that occurs more than once, either within a program or across different programs owned or maintained by the same entity*" [9][11][25][42].

Programmers are used some techniques for " Writing Code ". Once approaching " Writing Code ",it is important to mention one of the widely used techniques in this field known as : Copy and Paste . The majority of developers have been utilizing such a popular method for writing code due to a number of main advantages , most importantly being simple to use. Another major advantage is the fact that such a method is often used for less time-consuming and cost when developing the software system. In other words, a final software product is delivered in the shortest possible time with relatively minimum cost. which is regarded as one of the main reasons for such intentional clones that are beneficial in many ways [42].

2.4.1.1 The Origin of Duplicated Code (Clones)

Software reuse is the process of creating software systems from existing software rather than building everything from scratch. The kinds of artefacts that can be reused are not limited to source code fragments. They may include design structures, modulelevelimplementation structures, specifications, documentation, transformations, and more. Forms of code source reuse include loops, functions, procedures, subprograms ,subroutines, software component libraries, inheritance, application generators, generic software templates. The mechanisms of software reuse are well integrated in the software development process [9][41].

Many programmers rather adopt an apparently simpler approach to reusing software system designs and source code. They collect fragments from existing software systems and use them as part of new software by simply applying the well known practice that we call **copy-and-paste method**. This occurs frequently during the development phase when they reuse tried and tested code in a new context. Every developer copies pieces of software. When encountering a familiar problem that has been solved before, it's a normal reflex to reuse the existing code. One does not have to reinvent the wheel. This copy-and-paste programming style leads to duplicated code [9].

Duplication occurs also during the maintenance phase when the program must be adapted to the new requirements of the users: a program that is used in a real-world environment must be changed to add new functionality or to adapt to changes in the environment. Since the existing system already treats many problems of the domain, an obvious way to integrate the changes is to copy fragments with only small modifications. Duplicated code is therefore a phenomenon that occurs frequently in large systems. Some of reasons why programmers duplicate code are listed below[11][59]:

- The following conditions in the development environment can increase the trend to code duplication:
  1) There is no time to design, implement, and test a newly developed component. If a programmer cannot finish on time, it's wiser to copy a piece of code that runs properly than to persist in doing a very good design that will not run.
  2) Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price.
  3) The productivity of developers is sometimes measured in terms of number of lines of code written. This rewards copy-and-paste rather than writing new code.
- The programmer personality: we all have a natural laziness.
  1) Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already have been tested so the introduction of a bug seems less likely.
  2) Making code reusable takes extra efforts.

2.4.1.2 Classification of Duplicated Code

Similar or duplicated code fragments are known as code clones. Over more than a decade of research on code clones, the following categorising definitions of code clone have been widely accepted today [9][11][25][42].

- ✓ **Type-1 clones**: Identical code fragments except for variations in white-spaces and comments are 'Type-1' clones.
- ✓ **Type-2 clones**: Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments are called 'Type-2' clones.
- ✓ **Type-3 clones**: Code fragments that exhibit similarity as of 'Type-2' clones and also allow further differences such as additions, deletions or modifications of statements are known as 'Type-3' clones.
- ✓ **Type-4 clones**: Code fragments that exhibit identical functional behaviour but implemented through different syntactic structure are known as 'Type-4' clones.

Type-1' clones are also called 'exact' clones, whereas the 'Type-2' and 'Type-3' clones are also known as 'near-miss' clones. Owing to the semantic similarity rather than syntactic similarity, 'Type-4' clones are also referred to as 'semantic' clones. Our work deals with the exact (Type-1) and near-miss (Type-2 and Type-3) 'block' clones excluding the semantic ('Type-4') clones, because the accurate detection of semantic ('Type-4') clones is still an open problem [25].

Although copy-and-paste programming helps to meet short term goals (the code is already designed, implemented and debugged), it involves a lot of problems in software maintenance, which is estimated to cost 70% of the overall effort for producing software system in average[9]:

1) It complicates the comprehension of the program.
2) Code duplication increases the size of the code, extending compile time and expanding the size of the executability.

3) It uses more memory and complicates the error detection. Defects found in a code segment that has possibly been copied involves searching the clones of the segment and assessing the impact of the correction in each new context. If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked.
4) Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality.

2.4.2 The Problem of Long Method

One of easy smell to identify in a code is “Long Method”. Is a method that is too long, which contains too many lines of code. Methods that are longer than 10 lines are generally viewed as potential problem areas and can harm the readability and maintainability of your code [38][50].

Among all types of object oriented code, classes with short methods live longest. The longer a method or function is, the harder it becomes to understand and maintain it. In addition, long methods offer the perfect hiding place for unwanted duplicate code.

Since it is easier to write code than to read it, this "smell" remains unnoticed until the method turns into an ugly, oversized beast.

We want to improve the readability of this code by restructuring it. As has been mentioned above, it’s going to be attempted to decompose this method into smaller, more readable methods by using the “Extract Method” restructure.

Before we start, it’s a good idea to identify the behaviour of the method so that it can ensured that the code behaves the same after restructuring. It can often be tempting to add new functionality or to change how the code works but this can cause errors and should be resisted if possible.

2.4.3 The Problem of Large Class

When a class is trying to do too much or It's doing a lot of things, but it seems to have a variety of responsibilities. When a class has too many instance variables, fields, methods or lines of code, duplicated code cannot be far behind [60].

Classes usually start small. But over time, they get bloated as the program grows. As is the case with long methods as well, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.

2.5 Side Effects Of Code Smells

One of the most important negative effects of the smells in the source code is[17][23][83]:

2.5.1 Software Performance

Performance is an important quality attribute of software architecture. It can be characterized by metrics such as response, time, throughput, and resource utilization. In many existing systems, the reason for bad performance is a poorly code designed software architecture. According to Martin Fowler: "Code smells are usually not bugs and Smells do not prevent the current program from functioning. Instead, they indicate weaknesses in design that may consume computer resources"; Therefore, the smells negatively affect in the software performance. Performance predictions based on architectural descriptions of a software system can be performed before the implementation starts, which can possibly reduce cost for subsequent changes to fix performance problems. It is the hope that such early analyses support the decision for design alternatives and reduce the risk of having to redesign the architecture after performance problems have been diagnosed in the implementation [83].

Performance is critical to the success of today's software systems. However, many software products fail to meet their performance objectives when they are initially constructed. Fixing these problems is costly and causes schedule delays, cost

overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. In extreme cases, it may not be possible to fix performance problems without extensive redesign and re-implementation. In those cases, the project either becomes an infinite Consumer for time and money, or it is cancelled [52][36].

In the next section [Problems that appear in the system (2.6)] you will mention the remainder of the side effects of the code smells

2.6 Detection of Bad Smells

Robert C. Martin refers to "design smells" as higher-level smells that cause the decay of the software system's structure. He states they can be detected when software starts to exhibit the following problems[82].

- ✓ **Rigidity**: The design is hard to change because every change forces many other changes in other parts of the system.
- ✓ **Fragility**: The design is easy to break. Changes cause the system to break in places that have no conceptual relationship with the part that was changed.
- ✓ **Immobility**: It is hard to disentangle the system into components that can be reused in other systems.
- ✓ **Viscosity**: Doing things right is harder than doing things wrong. It is hard to do the right thing because sometimes it is just easier to do "quick hacks".
- ✓ **Needless Complexity**: The system is over-designed, containing infrastructure that adds no direct benefit.
- ✓ **Needless Repetition**: The design contains repeating structures that could be unified under a single abstraction.
- ✓ **Opacity**: The system is hard to read and understand and does not express its intent well.

2.7 The Problem

Duplicated Code, Large class and Long Method are phenomenon that occurs frequently in large systems for several reasons (see section 1.1). Although code duplication , large class and long method can have justifications, it is considered a bad practice. During maintenance, which is estimated at 70% of the overall effort for producing a software system, duplicated codes, large class and long method give the following problems [8][9][11][22]:

1. Hindrance to comprehension of the program.
2. Independent evolution of the clones.
3. Bad design.
4. suffer from quality problems with respect to internal quality aspects like usability, maintainability, or reusability.
5. To improve software design quality.
6. To increase understandability of the code.
7. To reduce project evolution time, especially in source code management activities.

In this thesis we investigate how this problems can be solved in software systems are developed using object oriented language.

2.8 Previous Work

Prajakta Ashtaputre and et al(June ,2016), proposed another different approach by refactoring opportunities for Detected Code Clones (duplication code) in source codes. This approach consists of two stages: The first one is to detect the clones which usually exist in the *source file or source code*. This also involves taking input as source file. The second step is to distinguish between refactorability and non-refactorability clones.

This approach " was able to check the refactorability opportunity for clone pair that is only TWU code fragments which are detected as clones "*(**Prajakta Ashtaputre and et al, June 2016).***

Fabio Palomba(May 2015), however. suggested different means to remove Smells, which in turn help the developer in program boosting. His approach known as *Textual Analysis Techniques* is to identify smells in source code." The proposed textual-based approach for detecting smells in source code, coined as TACO (Textual Analysis for Code smell detection), has been instantiated for detecting the *Long Method* smell and has been evaluated on three Java open source projects ". ***(Fabio Palomba, May 2015)***

Naouel Moha and et al(January/February, 2010), have provided a tool, called " DECOR:" Tool in the process of embodying and defining all the steps necessary for the specification and detection of code smells. These essential steps are developing to automate this process as much as possible. This tool runs in Java and is currently designed for Java legacy systems.

Whitfield and et al (November, 1997),proposed a framework that enables the exploration, both analytically and experimentally the properties of code-improving transformations. This framework includes a technique that facilitates an analytical investigation of code-improving transformations using the Gospel specifications, and contains a tool, called " ***Genesis***", that automatically produces a transformer that implements the transformations specified in Gospel.

CHAPTER 3
An Enhancement Approach of Software System

3.1 Introduction

The success of software system requires some factors, such as approach, methodologies, metrics, standardization in system design, and code, environment as well. This chapter, introduces Enhancement Approach for improving the quality of the software code, it simply "improving the design of existing code without changing its behaviour", it attempts to benefit from previous approaches to overcome code smell, which is considered as one of the existing risks, these smells may appear in the source code of software system. It is hoped that this approach is to be considered as the mainstream enhancement technique.

3.2 Smells Detection Technique or Process

Smells detection techniques can broadly be categorized as token-based, text-based, tree-based, graph-based, syntax-based, semantics-based, and metric-based, which have their advantages and weaknesses. For smells detection, this research suggest or adopt a hybrid approach combining strengths of multiple techniques; Moreover, these approaches depend on Graph-based technique with text-based technique to improve the precision of smells detection.

3.2 Constraints

The Enhancement Approach is based on two basic concepts: *Firstly*: The concept of situation is the basis of this approach: A set of applicable restructuring is associated to a given situation. The principle of this approach is to find for each smell situation and to propose a list of possible restructuring. *Secondly*: The concept of reverse engineering is the basis of this approach: Describing the software code using UML diagrams, In order to facilitate the process and identify the situations of each code smell. It is performed in two stages:

- ✓ Exploration and Assessment Stage.

- ✓ Code Restructuring Stage.

The following Figure presents an *abstract diagram* of the major stages for the process of smells detection used in the approach in hand:

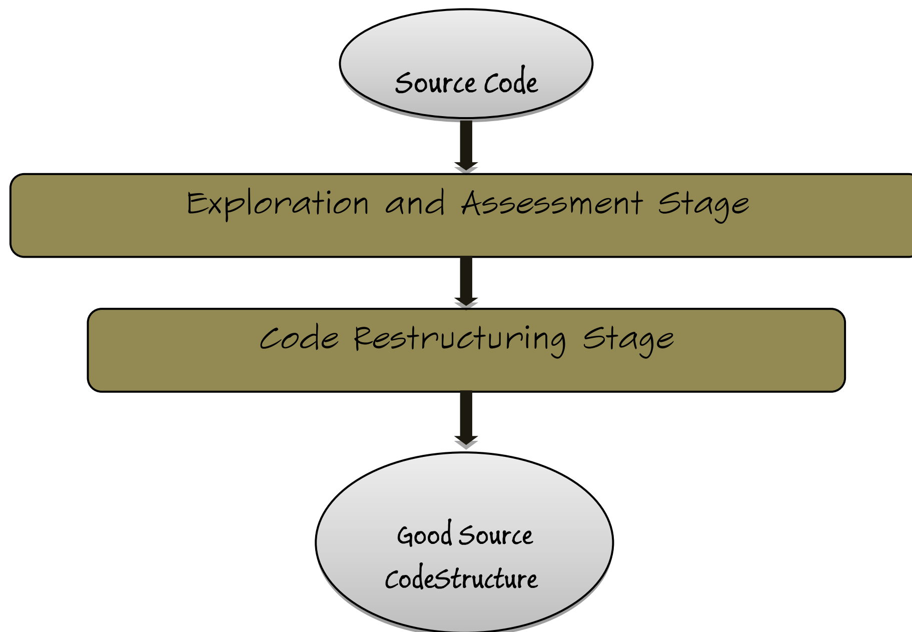


Figure 3.1: the Abstract Diagram of our Approach

Restructuring a software system means to refurbish it internally without interfering with its external behaviour. In other word, restructuring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. Firstly, an in-depth analysis are made which result in a list of findings. Ideally, these findings will be translated to UML diagrams, which define intermediate representation of software units that should be restructured. This exploration and assessment stage helps to understand the overall picture of the software project in mind and to divide it into several sub-projects, parts or restructuring units.

After that, a restructuring stage describes proposed solutions which are based on a contextual basis to tackle every problem individually. the end of this stage these solutions are applied in the source code. The following Figure presents a *schematic diagram* for the major modules for the source code enhancement process used in this approach:

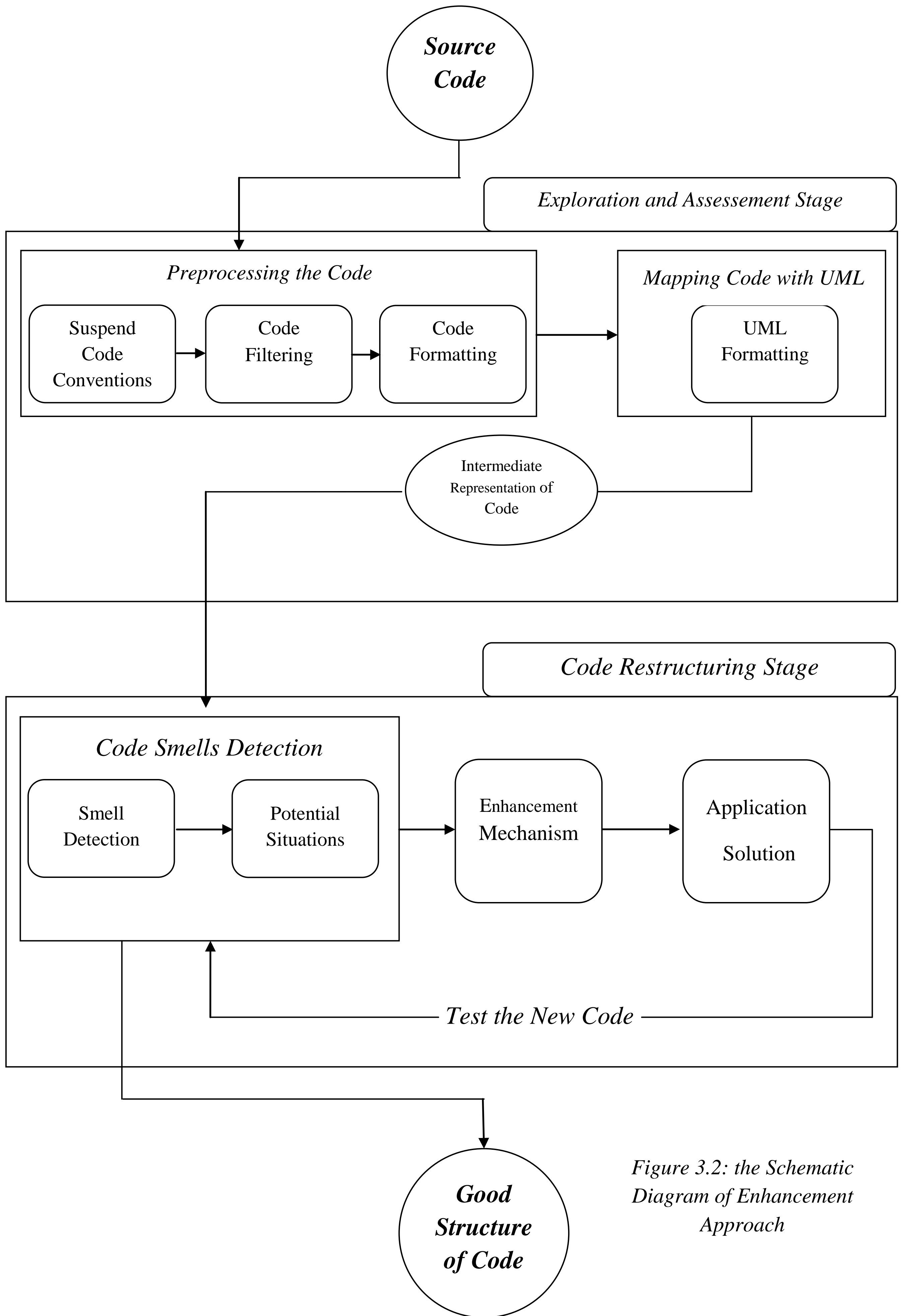


Figure 3.2: the Schematic Diagram of Enhancement Approach

3.3 An Enhancement Approach of Software System

Now, the basic idea of the approach in the context of smells detection and restructuring for systems written object oriented. It differs from the previous works in a general framework using suspend the code conventions, Code filtering, code formatting and mapping code with UML diagrams, determining of potential situations smells and execute appropriate restructuring. The above Figure, presents a schematic diagram for the major modules in the process of smells restructuring used in this approach.

This approach is called (Enhancement Approach for the Quality of Software Code), a loose name for accurate detection of three type of smells *(Duplicated code*, *Long method* and *Large class*). The main distinguishing characteristics of this method are the identification and extraction of the set of situations for each smell using UML diagrams and elimination of these situations that cause increasing the risk of bugs or failures in the future and consume computer resources, i.e. execution time and memory.

In the following section a detailed description of each step of the Enhancement Approach is to be provided:

First Stage: Exploration and Assessement Stage

The main goal of this stage is to understand and keep the overall picture of the software project that is to be restructured; in addition, an in-depth analysis of software systems are made, which result in suspending some and removing some unimportant material of the code (eg. whitespace, comments and others). And also this in-depth analysis result in dividing the large software system to several sub-projects known as restructuring units.

Once the restructuring units are determined, the source code of the restructuring units is transformed to an appropriate intermediate representation using UML Class diagram to restructure it. This transformation of the source code into an intermediate

representation is done by applying reverse engineering techniques. This stage is structured as follows:

First Preprocessing the Code(Preprocessing)

Before the restructuring units of code are transformed to an appropriate intermediate representation using UML Class diagram for restructuring it. The source code must pass through several phases, in order to guarantee that:

- ***First***, the source code is easily transformed to the intermediate representation.
- ***Second***, more importantly, the source code is partitioned and the domain of the comparison is determined.

In order to make the source code ready for transformation to an appropriate intermediate representation. There are three main phases that must be done; These phases are:

Phase 1 **Suspend Code Conventions**

In this phase, all the code conventions that exist in the source code are suspended. In fact, these code conventions are considered as one of very important part for the software system and also the software does not work without them. However, these parts are not important for the Enhancement Approach; in other words, these code conventions are not testable, but are suspend it at the beginning before the search of code smells starts in order to maintain these lines of code, which make them less subject to damage. After removing the code smells, the code conventions are returned.

The code conventions that should be suspended in the source code to complete the process of preparing code for transferring are to represent the medium representation are:

- Database declarations (the definitions or queries linked to the database). For example (e.g., SQL embedded in Java code).

- Library directives(statement that is used for defying library in system, such as #include).
- Some types of modifiers which are used to define some types of variables such as the final, and Constant.

Phase 1. I Methods of Suspend code conventions

This processing phase to suspend the conventions code is done by putting a mark (**) in front of the code lines that are used to link software system to the database, and also in front of the lines that are used to the database queries and declarations (e.g., SQL statement) in order to denote it. Moreover. for maintaining these lines of code from damage, lose or change. All of the above should be Suspend from the source code before proceeding to the next phase. The following Figure presents an example for the process of suspening conventions code in this approach.

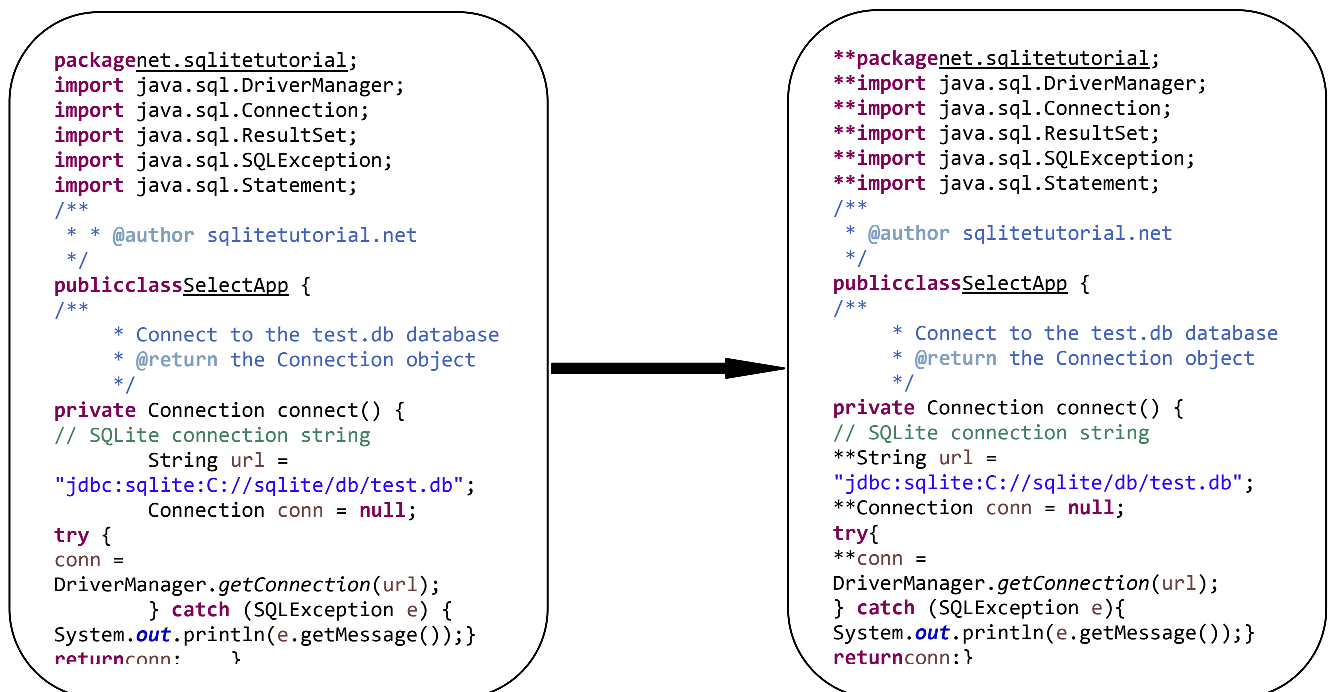


Figure 3.3: the process of remove (removal) uninteresting parts

Phase 2 **Code Filtering**

At this phase we remove all comments that are added by programmer in the source code. All comments are removed for three reasons:

- ✓ **First** : It reduces false positives by eliminating common constructs and idioms that should not be considered duplicated code. It also reduces false negatives by eliminating insignificant differences between software clones.
- ✓ **Second** :In order to guaranty, the run code lines are only existing in the source code, and if there is a similarity between code lines, then it is in the code line (duplicated code). In case the long function, the length is due to the number of run code lines and also in the case of the large class.
- ✓ **Third** : In order to reduce the run time of the software system. Because, the program compiler passes on all the code lines in the system; Even if it was, run code lines (performs a certain function) or explanatory comments that do not perform any function in the system. But, as soon as the compiler passes through this comments, the run time of software system are consumed; therefore, whenever the comments are more , they need the more time for compilation and run.

In this phase also, we remove the blanks in the source code. that helps us to find similarity between code lines, when we need to detection the duplicated code. Therefore, we remove the comments and blanks from each source code of software(e.g., methods, constructors or in the variables definition).

The tables below (2 and 3) present an example for the process of removal the blanks and comments in our approach.

Line Of Code	Original Code
1	`publicclassSelectApp`
2	`{`
3	`/**`
4	`     * Connect to the test.db database`
5	`     * @return the Connection object`
6	`     */`
7	`**private Connection connect()`
8	`**{`
9	`// SQLite connection string`
10	`**String url = **"jdbc:sqlite:C://sqlite/db/test.db";`
11	`**Connection conn = null;`
12	`**try`
13	`**{`
14	`**conn = **DriverManager.getConnection(url);`
15	`**}`
16	`        **catch (SQLException e)`
17	`**{`
18	`            System.out.println(e.getMessage());`
19	`**}`
20	`**returnconn;`
21	`**}`
22	`/**`
23	`/**`
24	`     * @param args the command line arguments`
25	`     */`
26	`*/`
27	`publicstaticvoid main(String[] args)`
28	` {`
29	`        SelectApp app = new SelectApp();`
30	`app.selectAll();`
31	`    }`
32	` }`
33	
Total of the number	
33	

table 2 Befor remove the blanks from the source code

Line Of Code	After filtering
1	`publicclassSelectApp{`
2	`**private Connection connect()**{`
3	`**String url = **"jdbc:sqlite:C://sqlite/db/test.db";`
4	`**Connection conn = null;`
5	`try{`
6	`**conn = **DriverManager.getConnection(url);    }`
7	`catch (SQLException e){`
8	`            System.out.println(e.getMessage());    }`
9	`returnconn; }`
10	`publicstaticvoid main(String[] args) {`
11	`        SelectApp app = new SelectApp();`
12	`app.selectAll();    } }`
Total of the number	
12	

table 3 After remove the blanks from the source code

In fact, there are some comments that are considered as important to the software the developers cannot dispense it, because it provides a very important explanation for the part of the source code (Clarify some complex functions) that is to be needed in the future (maintenance phase). These comments are temporarily deleted; In other word, after the approach apply is ended, these comments will be returned.

Phase 3 **Code Formatting**

The code formatting phase is considered as the last phase in the process of pre-processing the source code of software system. In this phase, the restructuring units of software system are determined. In order to transform it to an appropriate intermediate representation for restructure using UML Class diagram. This process is done as follows:

1. Each class of the software system, they are placed in a table that is represented a restructuring unit for this class; Moreover, the name table must be has the same name of the class and the unique number is given to this table.
2. The methods number constituent to the class are calculated. In order to help to determine the large class in the code smell detection phase later (see table 4).
3. The number of code lines for the each method are calculated. In order to help to determine the long method in the code smell detection phase later (see table 4).
4. If the class is related to another class or more, this relationships should be identified and mentioned as follows: Relationship type (Class _Name).

   Where: Class _Name: is the name of another class.

Note:

- ✓ *The interface in Java is treated like the class.*
- ✓ *The form in VB.Net is treated like the class.*

The following table (4) presents the example for the process of determining the restructuring units of the software system:

Line Of Code	Number Of Method line	Select App	Relationship	Class Number	Methods number
			Non	1	2
1		publicclassSelectApp{			
2	1/1	**private Connection connect()**{			
3	1/2	**String url = **"jdbc:sqlite:C://sqlite/db/test.db";			
4	1/3	**Connection conn = null;			
5	1/4	**try**{			
6	1/5	**conn = **DriverManager.getConnection(url); **}			
7	1/6	**catch (SQLException e)**{			
8	1/7	System.out.println(e.getMessage()); **}			
9	1/8	**returnconn; **}			
10	2/1	publicstaticvoid main(String[] args) {			
11	2/2	SelectApp app = new SelectApp();			
12	2/3	app.selectAll(); } }			

table 4 the determining the restructuring units of the system

Second Mapping Code with UML Diagrams

Once the code pre-processing stage is ended and the restructuring units are determined. In this phase, the source code of the restructuring units is transformed to an appropriate intermediate representation using UML Class diagram (UML Notation) for restructuring it. This intermediate representation is needed to bridge the gap between the Exploration and Assessment Stage and the code Restructuring stage because, Restructuring stage in our approach depends on UML Class diagram. It is simply not feasible to apply the best solution into a software system without this phase, because the Enhancement Approach is considered as hybrid. This transformation of the source code into an intermediate representation is done by applying reverse engineering concepts.

Secend. 1 **Reverse Engineering of Software**

The term reverse engineering as applied to software means different things to different people, prompting Chikofsky and Cross to write a paper researching the various uses and defining a taxonomy [5][6].

From their paper, they state, "*Reverse engineering is the process of analysing a subject system to create representations of the system at a higher level of abstraction*". It can also be seen as "*going backwards through the development cycle*". In this model, the output of the implementation phase (in source code form) is reverse-engineered back to the analysis phase, in an inversion of the traditional waterfall model. Another term for this technique is program comprehension. Reverse engineering, in computer programming, is a technique used to analyze software in order to identify and understand the parts it is composed of. The usual reasons for reverse engineering a piece of software is to recreate the program, to build something similar to it, to exploit its weaknesses and strengthen them [5][6][7][64].

Secend. 2 **Reasons for Reverse Engineering**

- ✓ *Lost documentation*: Reverse engineering often is done because the documentation of a particular device has been lost (or was never written), and the person who built it is no longer available. Reverse engineering of software can provide the most current documentation necessary for understanding the most current state of a software system [6].
- ✓ *Enhancement software product*: Some bad features of a product need to be eliminated e.g., excessive wear might indicate where a product should be improved. And also. Strengthening the good features of a product based on long-term usage [6].
- ✓ *Software modernization*: Often knowledge is lost over time, which can prevent updates and improvements. Reverse engineering is generally needed in order to understand the 'as is' state of existing or legacy software in order to properly estimate the effort required to migrate system knowledge into a 'to be' state. Much of this may be driven by changing functional, compliance or security requirements [6].
- ✓ *Bug fixing*: To fix (or sometimes to enhance) legacy software which is no longer supported by its creators (e.g. abandonware) [6].
- ✓ *Product analysis*: To examine how a product works, what components it consists of, estimate costs, and identify potential patent infringement [6].

Secend. 3 **Reverse Engineering Types**

In practice, two main types of reverse engineering emerge:[7][64]

In the first case: Source code is already available for the software, but higher-level aspects of the program, perhaps poorly documented or extracting the design diagrams.

In the second case: There is no source code available for the software, and any efforts towards discovering one possible source code for the software is regarded as reverse engineering. This second usage of the term is the one most people are familiar with.

However, in Enhancement Approach the first case of reverse engineering is used, because the source code is already available for the software system, but only want to strengthen and improve the source code by removing the code smells from them, in order to make the source code more understandable; Thus, the software system become more maintainable and maintenance cost is reduced.

Phase 4 **UML Formatting**

Steps to transform there structuring units to an appropriate intermediate representation using UML Class diagram in this approach:

1. Draw each class, interface or form (in general in UML) from the table that represent restructuring unit by using the UML class diagram with class/interface/form names only.

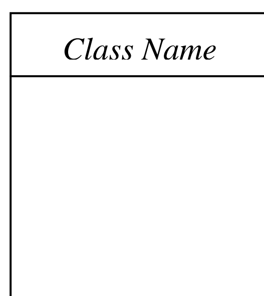


Figure 3.4: the UML class diagram that represent a class/interface/form.

2. Identify and draw *generalization (inheritance/extend)* relationship between two classes in the UML class diagram, from the relationship field in the table.
   - class A extends class {..........}

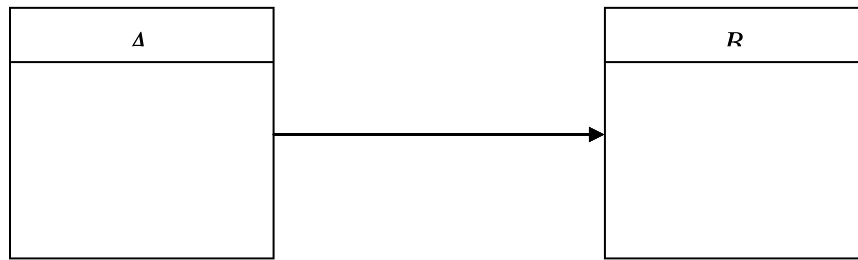


Figure 3.5: the generalization relationship between two classes/interfaces/forms.

3. Identify and draw *interface realization* (implement) relationship between a class and an interface in the UML class diagram, from the relationship field in the table.
   - class A implements IA {..........}

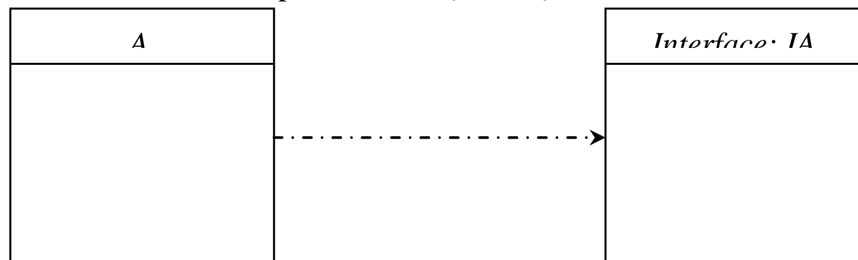


Figure 3.6: the interface realization relationship between a class and an interface

4. Identify and draw *directed association* relationship between two classes in the UML class diagram, from the relationship field in the table.
   - class A { private B b; …. }

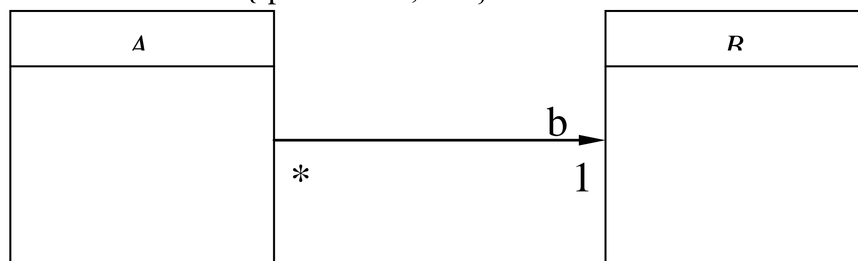


Figure 3.7: the directed association relationship between two classes.

5. Identify and draw instantiate dependency relationship between two classes from the relationship field in the table.
    - class A { … method(…) { … B b = new B(); … }
        - If there is already an A to B association relationship, do not draw instantiate relationship from A to B again.
        - Only consider creation/instantiation of long-lasting B object.

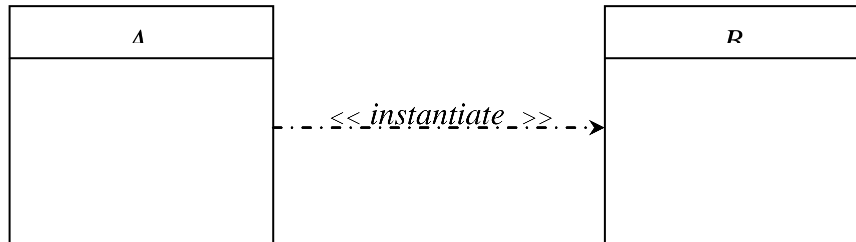


Figure 3.8: the instantiate dependency relationship between two classes.

6. Identify and draw *usage dependency* relationship between two classes in the UML class diagram, from the relationship field in the table.
    - class A { … method(B b) { … }

      or
    - class A { … method(…) { … B b; … }

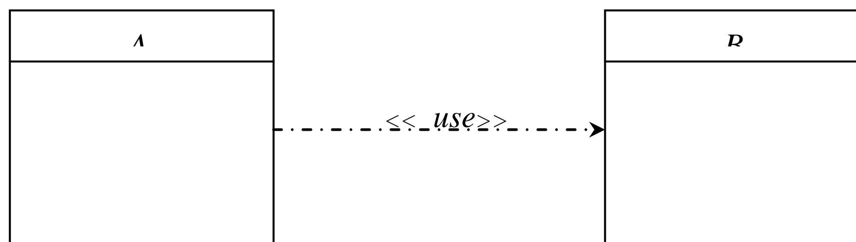


Figure 3.9: the usage dependency relationship between two classes.

7. Draw each methods in the class or interface (in general in UML)from the table that represent restructuring unit by using the UML diagram with method number and name of it only.

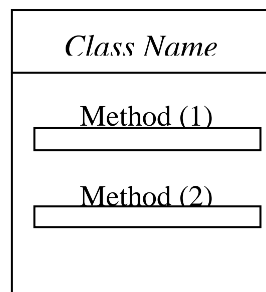


Figure 3.10: the methods in the class or interface.

The following Figure presents an example for the process of UML formatting used in our approach:

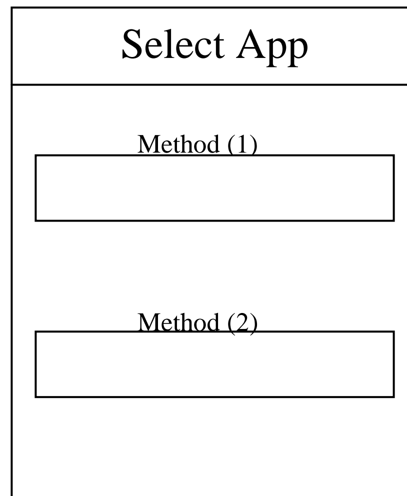


Figure 3.11: the example for the process of UML Formatting (transformation to an appropriate intermediate representation) used in the Enhancement Approach

Second Stage: ***Code Restructuring Stage***

Restructuring must be done systematically to reduce the risk of introducing bugs on the source code. Another equally important goal is to define the HOW, i.e. to find the best way to achieve the new design to avoid many risks. This includes to discover the order of the steps in which an actual restructuring can be made without breaking to much code at one time. Which starts from the clarification of way to detect smells, and also identify a set of situations are associated to a given smells. Then propose a list of possible restructuring for each situation. To do this, the phases listed later must be followed.

In fact, our approach (Enhancement Approach) is based on the concept of situation is the basis of the approach: is to find for each smell situation and to propose a list of possible restructuring. Obviously, during the assessment stage we gained a comprehensive understanding of the system architecture and this helps now to established the best practices for how to detect these smells and identify the situations that will be removed by proposed approach.

In the following phases a detailed description of each step of this process of detection of each smell consecutively is provided:

Phase 1 **Detection of Code Smells**

Detection of smells inside the source code of the system could be done in one of two ways, the first which is the easy one, by using some ready tools which is used for detection and analysis to indicate the place of code smells, however, one of the limitations of this way is to have to use more than one tool consecutively together in order to detect the three different types of smells, which our approach is treating and terminating them. Whereas the available tools are used to detect only one or two types at most of smells in the source code. So far there is not one tool that can deal with three smells. Most of the available tools detects the repetition of the code in the source code . one of the most common used tools to detect the duplication of the code is: Duploc; While as detection could be done the long method, large class by the following tools: Johnson and CC Finder.

The second way, which is more difficult, by using sight (observation) to detect smells through searching units of restructuring to detect the present smells in the source code. This is done through three main ideas to detect each smell in isolation.

The idea into understanding duplicated code, depending on finding the similarity in the code lines that create the source code system to find the duplicated code, either they were in the same method or in the different classes. also, the idea in long method is to know the number of lines that are found in the method. Where the methods that are longer than 10 lines are generally viewed as potential problem areas and can harm the readability and maintainability of the code. Then, the idea into knowing large class, depends on the class that contains the methods, which has been identified as the large class. Also, depending on the methods number which constitute the class. Where the class that has more than 10 methods is generally viewed as potential problem areas ,and can harm the readability and maintainability of the code.

After completion of smells detection inside the source code, then comes the process of defining which technique of restructuring is to be used to eliminate the

detected smells, In other words, it is decided which is the best technique of restructuring to be used to get rid of all types of smells. As in regard to the process of restructuring is considered a critical process which should be dealt with very carefully, a group of expected situations for each smell is to be defined, in order to facilitate restructuring through suggesting a list of solutions for each smell case. This facilitates choices for the developer to get rid of smells.

In the following sections, the situations described will be defined and it will be defined for each smell case. Each section contains a figure to illustrate the relation between classes in the system, a paragraph that describes the situation and a list of proposed restructuring.

Phase 1. I **The Potential Cases of Code Clone**

the idea in duplicated code is to define such situations, and to find out the corresponding set of restructuring. Depending on the relationship between classes containing the methods where the duplicated code was found, different situations that characterize them have been defined, and allow the definition of possible cures.

First Situation **Duplication in the Same Method**

Description: Two pieces of code (See Figure 3.12) are duplicated in the same method.

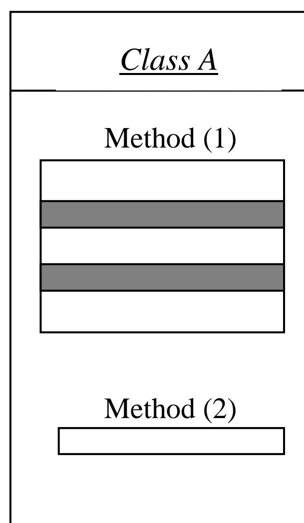


Figure 3.12: the duplication in the same method

Proposed Restructuring:

- Extract Method (See Section A.2).
- Parameterization (See Section A.3).

Discussion. This case represents the simplest situation. No attention should be paid to the side effects between classes. If an extract method is applied, the piece of code is replaced by a call to the newly created method. The signature of the original methods are not changed and a possible client does not see the difference. If there are no local variables the duplicated piece of code, it is propose at first the Extract Method restructuring for this situation. In some circumstance, the Parameterization could be used or a combination of both restructurings. The biggest problem with proposed restructuring , Extract Method, is dealing with local and temporary variables. In the simplest case, there is no local variable and the restructuring is trivially easy.

Second Situation **Duplication in the Same Class**

Description: Two different methods of the same class contain the same piece of code (See the following Figure 3.13).

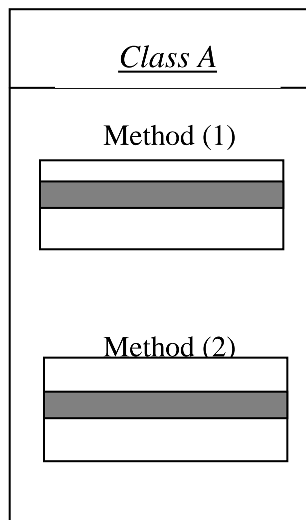


Figure 3.13: the duplication in the same class

Proposed Restructuring:

- Extract Method (See SectionA.2).
- Insert Method Call.
- Parameterization (See SectionA.3).
- Form Template Method (See SectionA.6).

Discussion: It is proposed four Restructuring which could be applied each alone or in combination. The Extract Method was discussed in the previous section. *Insert Method Call* could be applied when one method is entirely copied in the other method or when another method could be called with a special value.

Third Situation **Duplication between Sibling Classes**

Description: By sibling classes (See the following Figure3.14) we refer to all classes with the same direct superclass and with the same hierarchical level. The highlighted rectangles represent the classes in which the methods containing duplicated code were found.

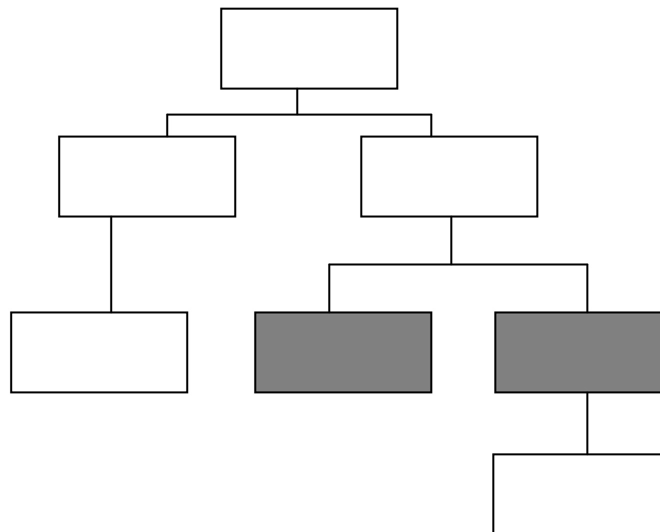


Figure 3.14: the duplication between sibling classes

Proposed Restructuring:

- Pull Up Method (See Section A.4).
- Parameterization (See Section A.3).

- Extract Method (See Section A.2).
- Substitute Algorithm (See Section A.7).
- Form Template Method (See Section A.6).
- Replace Subclass with Field (See SectionA.12).
- Extract Super-Class (See Section A.9).

Discussion: The experiments leads in previous works show the trend to pull up into the superclass the extracted duplication by using Form Template Method and Pull Up Method.

Fourth Situation   **Duplication with Super Class**

Description: This situation describes a duplication between a class and its direct superclass(See Figure 3.15).

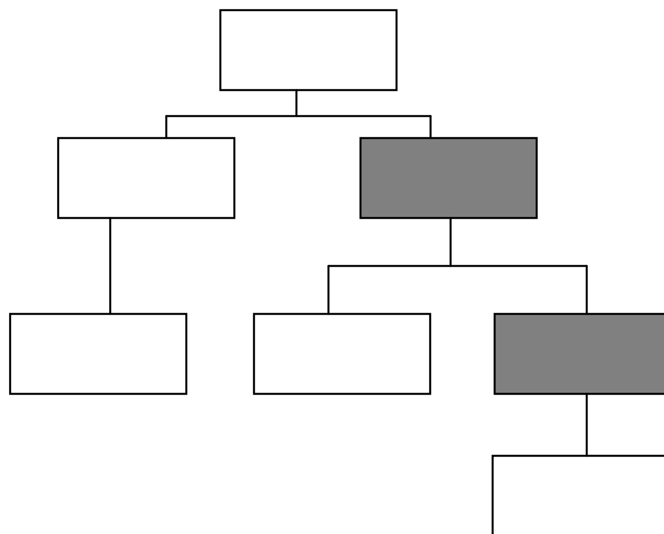


Figure 3.15: the duplication with superclass

Proposed Restructuring:

- Parameterization (See Section A.3).
- Insert Super Call.
- Pull Up Method (See Section A.4).
- Push Down Method (See Section A.5).
- Form Template Method (See Section A.6).

Discussion: If both methods have the same name, we can think on a template method for the restructuring or the duplication could also be eliminated by extracting method from both classes and then by putting it into the superclass.

Fifth Situation **Duplication with Ancestor**

Description: This situation describes the case where one class inherits from the other but not directly (See Figure 3.16).

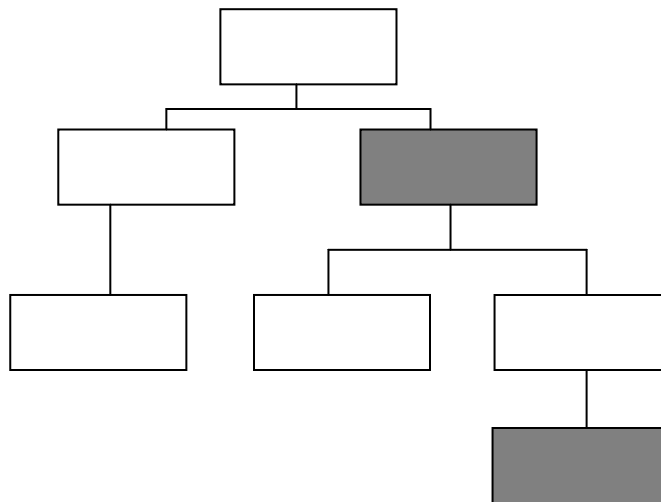


Figure 3.16: the duplication withancestor

Proposed Restructuring:

- Extract Method (See Section A.2).
- Parameterization (See Section A.3).
- Pull Up Method (See Section A.4).
- Form Template Method (See Section A.6).

Discussion: The difference to the previous situation(with superclass) is that if something is modified in the ancestor, all classes between the ancestor and the concerned subclass are also affected by the change. Vigilance must be there, for where the new created method is defined. If it is put it into the ancestor class, more classes are affected than in the case of superclass situation.

Sixth Situation **Duplication with First Cousin**

Description: This situation describes the case where both classes have the same hierarchical level and their super classes are sibling classes (See the Following Figure 3.17).

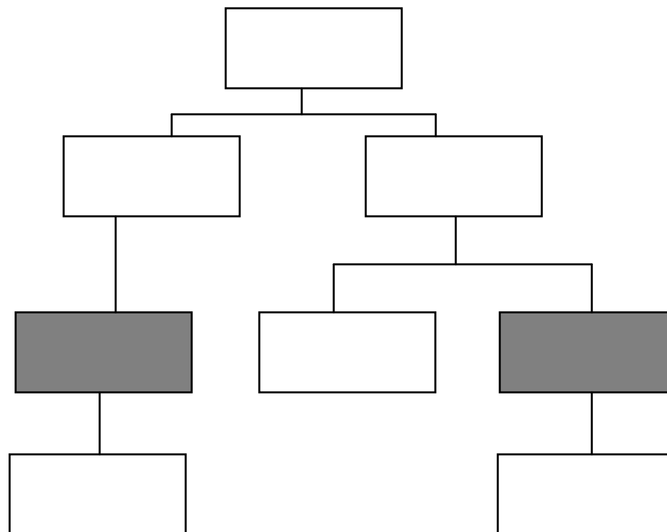


Figure 3.17: the duplication with first cousin

Proposed Restructuring:

- Pull Up Method (See Section A.4).
- Form Template Method (See Section A.6).
- Extract Method (See Section A.2).
- Parameterization (See Section A.3).
- Extract Super-class (See Section A.9).

Discussion: Using inheritance it can also pull up the extracted method two levels upper in the hierarchy. other classes with the same ancestor involved in the duplication must checked. If yes, “a flawed design” is a probability. All subclasses containing the same code may need a common superclass (Extract Superclass). One possibility is to extract a new superclass up to all concerned classes and to be put into it the new created component.

Seventh Situation **Duplication in Unrelated Classes**

Description: This situation describes the case where both classes do not have any common ancestor (See the Following Figure 3.18).

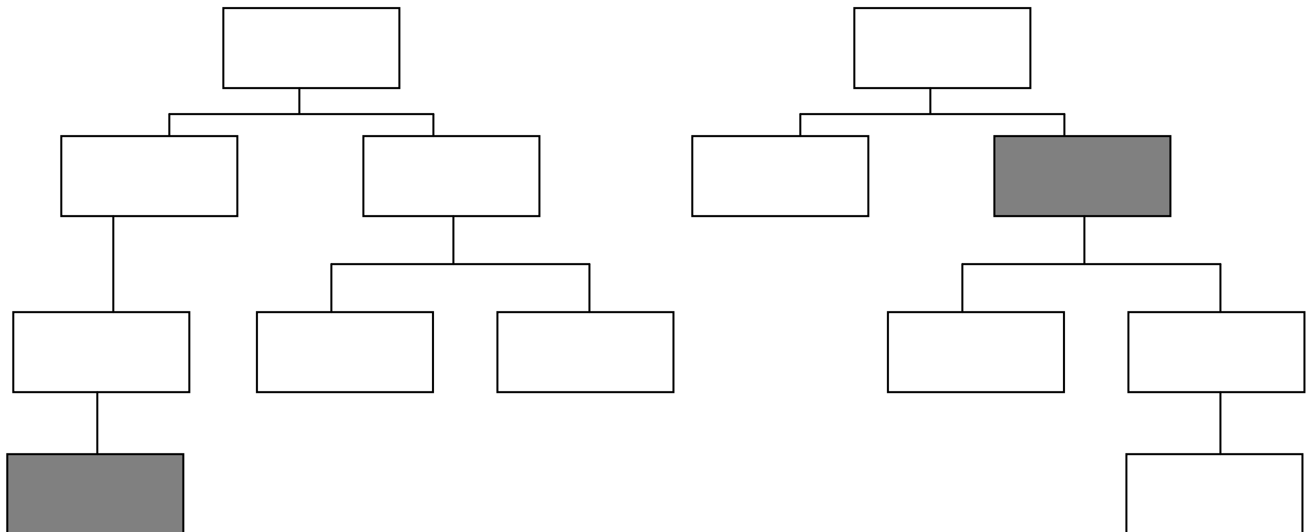


Figure 3.18: the duplication in Unrelated Classes

Proposed Restructuring: Proposing a solution for this situation is the most difficult one. If you have duplicated code in two unrelated classes, consider extracting a class from one class and then use the new component. If the method really belongs only to one of the classes, the other class should invoke it. You have to decide where the method makes sense and ensure it is there and nowhere else.

Phase 1. II **The Potential Cases of Long Method**

The idea in long method is to define such situations, and to find out the corresponding set of restructuring. Depending on the number of method lines contained in the class. Where the methods that are longer than 10 lines are generally viewed as potential problem areas and can harm the readability and maintainability of the code. different situations have already been defined that characterize the situation and allow the definition of possible cures.

First Situation **Long Method with More than 10 Lines**

Description: This situation describes the case where the method has a group of lines (more than 10 lines) or as little as a single line (even a single line) of code (See the Sollowing Figure 3.19).

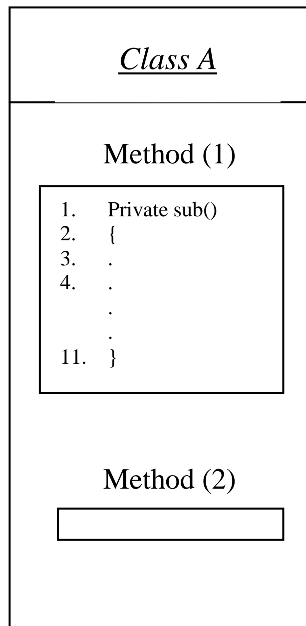


Figure 3.19: the long method with more than 10 lines

Proposed Restructuring:

- Extract Method (See Section A.2).
- Parameterization (See Section A.3).

Discussion: This case represents the simplest situation. No attention should be paid to side effects between classes. If an extract method is applied, the piece of code is replaced by a call to the newly created method. The signature of the original methods are not changed and a possible client does not see the difference. If local variable in the long method does not exist , it is proposed at first the Extract Method restructuring for this situation.

Second Situation **Long Method because of Duplicate Lines**

Description: This situation describes the case where two pieces of code (See Figure 3.20) are duplicated in the method. This situation is treated as (Duplication in the Same Method situation).

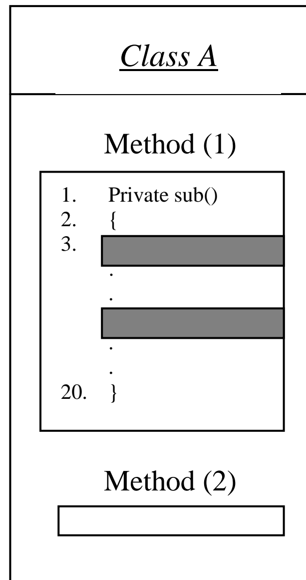


Figure 3.20: the long method because duplicate lines

Proposed Restructuring:

- Extract Method (See Section A.2).
- Parameterization (See Section A.3).

Discussion. this discussion is mentioned above (*First situation*. long method with more than 10 lines).

Third Situation **Long Method with Loops**

Description: This situation describes the case where the method has loops and the lines of code more than 10 lines (See the Following Figure 3.21).

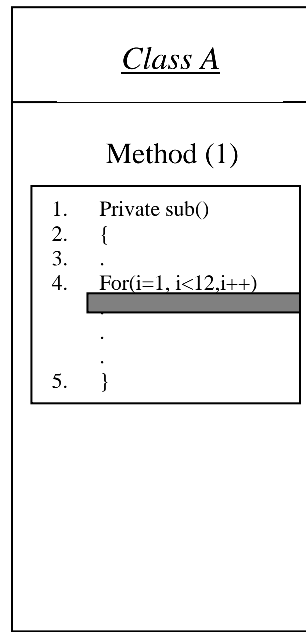


Figure 3.21: the long method with loops

Proposed Restructuring*:*

- Extract Method (See Section A.2).
- Parameterization (See Section A.3).

Discussion: This case represents the simplest situation. When the method is long (has more than 10 lines) and has a loop in its code. Extract the loop and the code within the loop into its own method.

Fourth Situation **Long Method with Conditional Expressions**

Description: The method containing the conditional expressions of code and is considered as long method (See the Following Figure3.22).

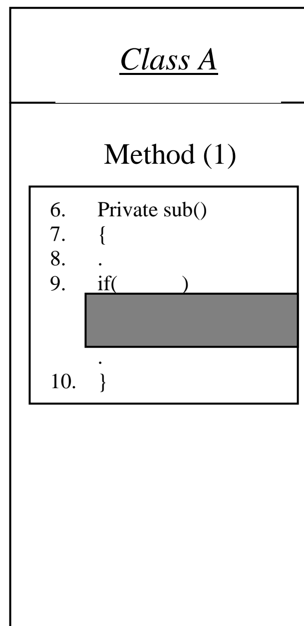


Figure 3.22: the long method with conditional expressions

Proposed Restructuring:

- Decompose Conditional (See Section A.13).

Discussion: This case represents the simplest situation. When the method is long (has more than 10 lines) and has a conditional expressions in its code. In this case Use Decompose Conditional to deal with conditional expressions.

Phase 1. III **The Potential Cases of Large Class**

the idea of a large class is to define such situations, and to find out the corresponding set of restructuring. Depending on the class that contains the methods, which has been identified as the long method. And also depending on the method's number which constitute the class, different situations have been defined that characterize the situation and allow the definition of possible cures.

First Situation **Large Class with Many Methods**

Description: When a class is trying to do too much; In other words, A class contains many methods of code. But over time, they get bloated as the program grows (See the Following Figure3.23).

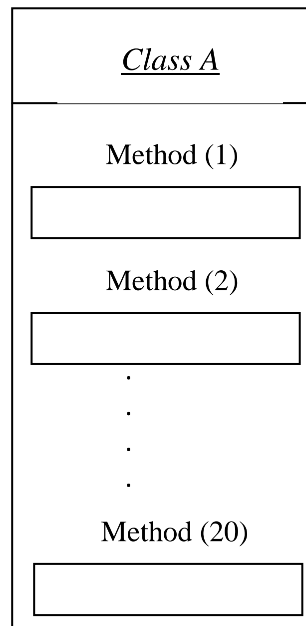


Figure 3.23: the large class with many methods

Proposed Restructuring:

- Extract Class (See Section A.8).
- Extract Subclass (See Section A.11).
- Extract Interface (See Section A.10).
- Extract Method (See Section A.2).

Discussion: Extracted Class helps if part of the behaviour of the large class can be spun off into a separate component. Extracted Subclass helps if part of the behaviour of the large class can be implemented in different ways or is used in rare cases. Extracted Interface helps if it is necessary to have a list of the operations and behaviours that the client can use.

Phase 2 **Enhancement Mechanism**

After the detection of the smells in earlier phases, and defining the expected solutions for each smell, in this phase the best choices and executions are to be chosen for the best solution for restructuring , the best solutions suggested to each smell depending on the case which appeared in the source code. This process should be conducted systematically to avoid or reduce entering errors in the source code. Therefore, a template was suggested to execute the process of smell reconstructing in an organized and simple way. Restructuring must be done systematically to avoid or reduce the risk of introducing bugs on the working code.

Each restructuring in this approach is implemented according to the following template(known as a restructuring template):

Phase 2. I **Restructuring Template**

1. **Bad smell**: The name of a bad smell situation.
2. **Method Name**: The name of a model restructuring Process.
3. **Location**: The restructuring area(s) of the transformation**.**
4. **Reasons :**Probable reason(s) for performing the restructuring.
5. **Description:** A short explanation of things if its intent is not obvious.
6. **Restructuring Process**: A mechanics of the improvement–identification of basic operations and/or other restructurings and the order in which they should be applied to achieve the objectives of the process; Hence, the Mechanics(Processes) that are used in this approach to the restructuring the source code are listed in **Appendix A**.

Phase 3 **Application Solution**

After the enhancement, the result will be obtaining a source code without these smells. It will carried out to make sure it excutes its functions. In this phase, code conventions (the code lines that are used to link software system to the database) are released by removing the mark (**) that are put in front of these code lines.

Moreover, some important developers comments that have been deleted from the source code are to be returned.

At the end of this phase, operating program, testing its functions to ensure that restructuration is perfectly done without impacting the system behaviour.

The main goal of this work is to manage the growth in size and complexity of a software system due to source code smell. For increase reliability, longevity and modifiability of software system. A large number of software smells induces undesirable side effects in a software system. The first possible effect is an increase in the resources required by the software on the system. This increases the cost of operation. The enhancement approach presents the way of eliminating these smells.

3.4 Summary

	Duplicated Code						
	Same Method	**Same Class**	**between Sibling Classes**	**with Superclass**	**with Ancestor**	**with First Cousin**	**Unrelated Classes**
Extract Method	√	√	√		√	√	
Parameterization	√	√	√	√	√	√	
Insert super Call				√			
Insert Method Call		√		√			
Form Template Method		√	√	√	√	√	
Pull Up Method			√	√	√	√	
Substitute Algorithm			√				
Replace Subclass with Field			√				
Extract Superclass			√			√	
Push Down Method				√			

table 5 the summary of code duplication restructuring mechanisms

	Long Method				*Large Class*
	more than 10 lines	**duplicate lines**	**with loops**	**conditional xpressions**	**many methods**
Extract Method	√	√	√		√
Parameterization	√	√	√		
Decompose Conditional				√	
Extract Class					√
Extract Subclass					√
Extract Interface					√

table 6 the summary of long method and large class restructuring mechanisms

CHAPTER 4
Case Study

4. 1 Introduction

This chapter briefly illustrates how Enhancement Approach can be applied, through a case study of the reinforcement of a code for General Mills Company system. By describing the stages they are followed for reinforcement of the Microsoft Visual Basic.Net 2010 system and the techniques that are used to detect code smells. It also explains how the techniques are used. For reasons of brevity, some details are omitted, that aim to give a general flavor of the improvement. Finally, the activities regarding Mills are evident in our case study. The actual implementation of the proposed solution to improve source code of the General Mills Company system is explained.

The following activities regarding General Mills Company system are evident in the case study. This system contains three classes; Each class must be isolated from the other classes but certainly dealing with each other to perform system functionality, and every class performs a set of special functions which are required.

The Enhancement Approach contains two basic Stages. Each Stage contains a set of phases. A detailed description of two stages and their phases depending on their effective application are provided:

First Stage: Exploration and Assessement Stage

The target of this stage is to make (performance) an in-depth analysis of software system that is needed to be restructured, which result to remove some unimportant things of the code. After that, restructuring units are identified in order that the source code is partitioned to facilitate the determination of the comparison domain. At the end of this stage, using the reverse engineering concepts specifically the class diagram. The restructuring units are transferred to the intermediate representation.

The actual implementation of this stage is in the following phases:

First Preprocessing The Code(Preprocessing)

In order to make the source code possible to transferred to the intermediate representation, The existence of code conventions, comments and blanks must be removed from the source code; For doing this, you must follow these phases:

Phase 1 **Suspend Code Conventions**

Now, all the code conventions are suspended out in the source code; that is done by putting a mark (**) in front of the code conventions. This phase will be applied to each source code for the case study. The following figure (the first) shows a part of this actual Apply:

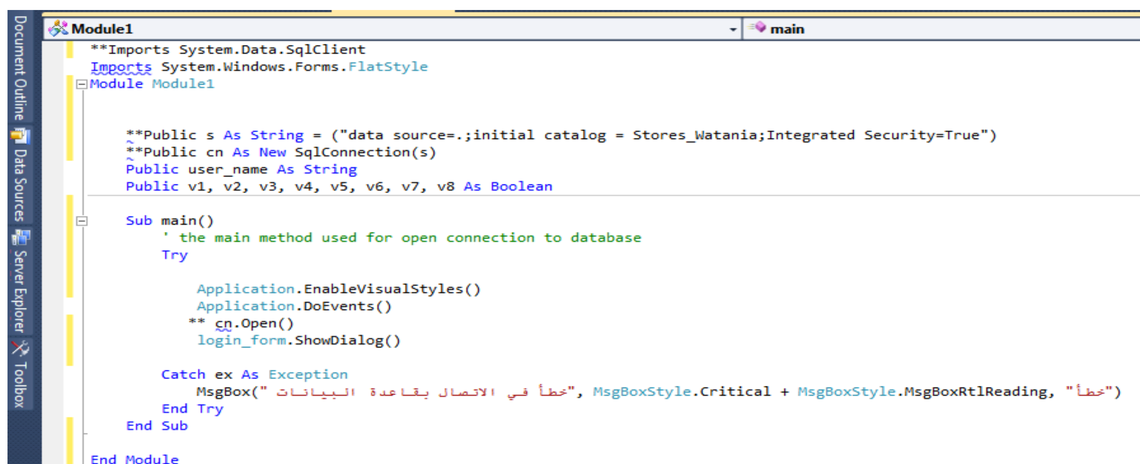


Figure 4.24: the code conventions are suspended

Phase 2 **Code Filtering**

At this phase; All the blanks between the source code lines of the system that are added when the code is wrote to facilitate the process of separation between source codes are removed. That is to be applied to each source code for the case study. The following table (7) shows a part of this actual apply:

Original Code

<pre>PrivateSub Button7_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button7.Click
'**
****
'حذفاذنالاتلافمنلستفيو الثانية
 	
ForEach item In ListView2.SelectedItems
            ListView2.Items(item.Index).Remove()
 	
Next
'
 	
        ListView2.BackColor = Color.White
Dim col AsInteger = 0
 	
For i = 0 To ListView2.Items.Count - 1
If col Mod 2 = 0 Then
                ListView2.Items.Item(i).SubItems(0).BackColor =
Color.LightGoldenrodYellow
Else
                ListView2.Items.Item(i).SubItems(0).BackColor = Color.White
EndIf
            col = col + 1
Next
 	
'@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@22
'clear1
        TextBox9.Clear()
        TextBox7.Clear()
        Label13.Text = ""
        Button7.Enabled = False
EndSub</pre>

Total of the number
30

table 7 before remove the blanks from the source code

Note that the source code of the method starts from 1 to 30; Therefore, line of code (LOC) for method is (30). And once the blanks and blocks are deleted the LOC of this method is reduced to (6). In addition, delete blanks process makes method have only been contained to source code.

After the completion of the blanks removal process. Now, all comments are to be removed that are added by programmer in the source code. These comments are removed because it reduces false positives by eliminating common constructs and

idioms that should not be considered duplicated code. It also reduces false negatives by eliminating insignificant differences between software clones. This phase will be applied to each source code for the case study. The following figure (8) shows a part of this actual Apply:

After filtering
(code below)
Total of the number
19

```
PrivateSub Button7_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button7.Click
ForEach item In ListView2.SelectedItems
            ListView2.Items(item.Index).Remove()
Next
        ListView2.BackColor = Color.White
Dim col AsInteger = 0
For i = 0 To ListView2.Items.Count - 1
If col Mod 2 = 0 Then
                ListView2.Items.Item(i).SubItems(0).BackColor =
Color.LightGoldenrodYellow
Else
                ListView2.Items.Item(i).SubItems(0).BackColor = Color.White
EndIf
            col = col + 1
Next
        TextBox9.Clear()
        TextBox7.Clear()
        Label13.Text = ""
        Button7.Enabled = False
EndSub
```

table 8 After remove the blanks from the source code

Now note that, the code line of this method becomes the shortest after the comments are deleted; LOC for this method is (19). This makes, the comparison process between the code lines very easy.

In fact, there are some comments that are considered as important to the software developers and the developers cannot dispense it, because it provides a very important explanation for the part of the source code (i.e... Method ()) that we are needed it in future (maintenance phase). this comment is temporarily deleted; In other words, after the approach apply is ended, these comments will be returned.

Phase 3 **Code Formatting**

The code formatting phase is the last phase in the pre-processing process of the source code of software system. Now, the source code is ready for determining its restructuring units.

In this phase, the software system restructuring units are determined. The process is applied or done to the proposed system (case study) as follows:

Number Of Method line	*Main*	*Relationship*	*Class Number*	*Methods number*
		• **Module1** • **Users** • **Edin_Estelam_bill** • **Edn_srf_bill** • **Edn_Etlaf_bill** • **Suppliers** • **Store** • **Login_form** • **Card_items**	**2**	**17**

```
1.   **Imports System.Data.SqlClient
2.   PublicClassmain
1/1       Sub User_rights()
1/2       m1.Enabled = v1
1/3       m2.Enabled = v2
1/4       m3.Enabled = v3
1/5       f1.Enabled = v4
1/6       f2.Enabled = v5
1/7       f3.Enabled = v6
1/8       i.Enabled = v7
1/9       u.Enabled = v8
1/10      EndSub
2/1       PrivateSub Timer1_Tick(ByVal sender As System.Object, ByVal e As
          System.EventArgs) Handles Timer1.Tick
2/2       ToolStripStatusLabel1.Text = "التاريخ:     " + Format(Now,
          "yyyy/MM/dd") + "               "
2/3       ToolStripStatusLabel3.Text = "الوقتالحالي:     " + Format(Now,
          "hh:mm:ss tt")
2/4       EndSub
3/1      PrivateSub main_Load(ByVal sender As System.Object, ByVal e As
         System.EventArgs) HandlesMyBase.Load
3/2      User_rights()
3/3      ToolStripStatusLabel5.Text = "المستخدمالحالي: " + user_name
3/4      EndSub
4/1      PrivateSub m10_Click(ByVal sender As System.Object, ByVal e As
         System.EventArgs)
4/2      users.ShowDialog()
4/3      EndSub
```

```
5/1     PrivateSub e1_Click(ByVal sender As System.Object, ByVal e As
        System.EventArgs) Handles e1.Click
5/2     Dim ms AsString
5/3     ms = MsgBox("هلبالتأكيدتريدالخروجمنالنظام", MsgBoxStyle.YesNo +
        MsgBoxStyle.Question + MsgBoxStyle.MsgBoxRight +
        MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
5/4     If ms = vbYes Then
5/5     Dispose()
5/6     EndIf
5/7     EndSub
6/1     PrivateSub e2_Click(ByVal sender As System.Object, ByVal e As
        System.EventArgs) Handles e2.Click
6/2     Dim ms AsString
6/3     ms = MsgBox("هلبالتأكيدتريدتبديلالمستخدم", MsgBoxStyle.YesNo +
        MsgBoxStyle.Question + MsgBoxStyle.MsgBoxRight +
        MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
6/4     If ms = vbYes Then
6/5     Me.Dispose()
6/6     login_form.ShowDialog()
6/7     EndIf
6/8     EndSub
7/1     PrivateSub u2_Click(ByVal sender As System.Object, ByVal e As
        System.EventArgs) Handles u2.Click
7/2     Try
7/3     **Dim sqlConn AsNewSqlConnection("data source=.;Initial catalog =
        Stores_Watania;Integrated Security=True")
7/4     **sqlConn.Open()
7/5     **Dim sCommand = "BACKUP DATABASE [Stores_Watania] TO DISK =
        N'd:\backup Stores_Watania.bak' WITH COPY_ONLY"
7/6     **Using sqlCmd AsNewSqlCommand(sCommand, sqlConn)
7/7     **sqlCmd.ExecuteNonQuery()
7/8     **sqlConn.Close()
7/9     EndUsing
7/10    **Dim co AsString = "النسخةالاحتياطيةلقاعدةبياناتمخازنشركةالمطاحنالوطنية.bak"
7/11    SaveFileDialog1.Title = "النسخالاحتياطي"
7/12    SaveFileDialog1.Filter = "قاعدةالبيانات|*.bak"
7/13    SaveFileDialog1.FileName = co
7/14    If SaveFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
7/15    IO.File.Copy("d:\backup Stores_Watania.bak",
        SaveFileDialog1.FileName, True)
7/16    MessageBox.Show("تمتنجاحعمليةالنسخالإحتياطي")
7/17    EndIf
7/18    Catch ex AsException
7/19    MsgBox("يجباعادةالعملية", MsgBoxStyle.Information +
        MsgBoxStyle.MsgBoxRight + MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
7/20    EndTry
7/21    EndSub
8/1     PrivateSubمعدالنظامToolStripMenuItem_Click(ByVal sender As
        System.Object, ByVal e As System.EventArgs)
        HandlesمعدالنظامToolStripMenuItem.Click
8/2     system_by.ShowDialog()
8/3     EndSub
9/1     PrivateSub m1_Click(ByVal sender As System.Object, ByVal e As
        System.EventArgs) Handles m1.Click
9/2     Edin_Estelam_bill.ShowDialog()
9/3     EndSub
10/1     PrivateSub m2_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles m2.Click
10/2     Edn_Srf_bill.ShowDialog()
10/3     EndSub
11/1     PrivateSub u1_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles u1.Click
```

``` 11/2 users.ShowDialog() 11/3 EndSub 12/1 PrivateSub i1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles i1.Click 12/2 bill_reports.ShowDialog() 12/3 EndSub 13/1 PrivateSub m3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles m3.Click 13/2 Edn_Etlaf_bill.ShowDialog() 13/3 EndSub 14/1 PrivateSub f1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles f1.Click 14/2 Suppliers.ShowDialog() 14/3 EndSub 15/1 PrivateSub f2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles f2.Click 15/2 Card_items.ShowDialog() 15/3 EndSub 16/1 PrivateSub f3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles f3.Click 16/2 store.ShowDialog() 16/3 EndSub 17/1 PrivateSub i2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles i2.Click 17/2 reorder_point_report.ShowDialog() 17/3 EndSub 90. EndClass ```	
Total of LOC	***90***

table 9 the determining the restructuring unit for the Main form

Likewise, this process will be applied to all classes and forms of the software system, but because the case study has many classes and forms(eight forms and three classes) these a very large number of forms, in order to simplify the purpose of non-lengthening, the rest of the restructuring units will be determinate for the software system and without the source code here.

Number Of Method line	***Card _items***	***Relationship***	***Class Number***	***Methods number***
		• **Module1**	**3**	**11**

```
1. **Imports System.Data.SqlClient
2. PublicClassCard_items
3. **Inherits System.Windows.Forms.Form
4. .
5. .
   .
   .
   .
```

234. EndClass	
Total of LOC	***234***

table 10 the determining the restructuring unit for the Card _items form

Number Of Method line	***Edn_Etlaf_bill***	***Relationship***	***Class Number***	***Methods number***
		1. Module1	**4**	**22**

```
1. **Imports System.Data.SqlClient
2. Imports System.Data
3. PublicClassEdn_Etlaf_bill
4. .
5. .
   .
   .
   .
489.      EndClass
```

Total of LOC	***489***

table 11 the determining the restructuring unit for the Edn _Etlaf _bill form

Number Of Method line	***Edn_Srt_bill***	***Relationship***	***Class Number***	***Methods number***
		1. Module1	**5**	**23**

```
1. **Imports System.Data.SqlClient
2. Imports System.Data
3. PublicClassEdn_Srf_bill
4. .
5. .
   .
   .
569.      EndClass
```

Total of LOC	***569***

table 12 the determining the restructuring unit for the Edn _Srt_ bill form

<table>
<tr><td rowspan="2">Number Of Method line</td><td rowspan="2">Loing_form</td><td>Relationship</td><td>Class Number</td><td>Methods number</td></tr>
<tr><td>2. Module1
3. main</td><td>6</td><td>5</td></tr>
<tr><td colspan="5">

```
1. **Imports System.Data.SqlClient
2. PublicClasslogin_form
3. Inherits System.Windows.Forms.Form
4. .
5. .
   .
   .
132.      EndClass
```

</td></tr>
<tr><td>Total of LOC</td><td colspan="4">132</td></tr>
</table>

table 13 the determining the restructuring unit for the Loing_form form

<table>
<tr><td rowspan="2">Number Of Method line</td><td rowspan="2">Store</td><td>Relationship</td><td>Class Number</td><td>Methods number</td></tr>
<tr><td>4. Module1</td><td>7</td><td>11</td></tr>
<tr><td colspan="5">

```
1. **Imports System.Data.SqlClient
2. PublicClassstore
3. Inherits System.Windows.Forms.Form
4. .
5. .
   .
   .
286.      EndClass
```

</td></tr>
<tr><td>Total of LOC</td><td colspan="4">286</td></tr>
</table>

table 14 the determining the restructuring unit for the store form

<table>
<tr><td rowspan="2">Number Of Method line</td><td rowspan="2">Suppliers</td><td>Relationship</td><td>Class Number</td><td>Methods number</td></tr>
<tr><td>5. Module1</td><td>8</td><td>13</td></tr>
<tr><td colspan="5">

```
1.  **Imports System.Data.SqlClient
2.  PublicClassSuppliers
3.  Inherits System.Windows.Forms.Form
4.  .
5.  .
    .
    .
233.      EndClass
```

</td></tr>
<tr><td>Total of LOC</td><td colspan="4">233</td></tr>
</table>

table 15 the determining the restructuring unit for the suppliers form

<table>
<tr><td rowspan="2">Number Of Method line</td><td rowspan="2">Users</td><td>Relationship</td><td>Class Number</td><td>Methods number</td></tr>
<tr><td>6. Module1</td><td>9</td><td>15</td></tr>
<tr><td colspan="5">

```
1.  **Imports System.Data.SqlClient
2.  PublicClassusers
3.  Inherits System.Windows.Forms.Form
4.  .
5.  .
    .
    .
534.      EndClass
```

</td></tr>
<tr><td>Total of LOC</td><td colspan="4">534</td></tr>
</table>

table 16 the determining the restructuring unit for the users form

Now, the eleven restructuring units for the software system were clearly and accurately identified; also, the software system is ready for transform it to an appropriate intermediate representation.

Second Mapping Code with UML Diagrams

Phase 4 UML Formatting

In this section, the transformation of an appropriate intermediate representation using the Project Analyst application (Version 10.2) will be discussed. One of the advantages of this application is that it supports the concepts of the Reverse Engineering that have been used in the Enhancement Approach for the initial description of the classes and their relationship with each other. The following figure represents the UML formatting:

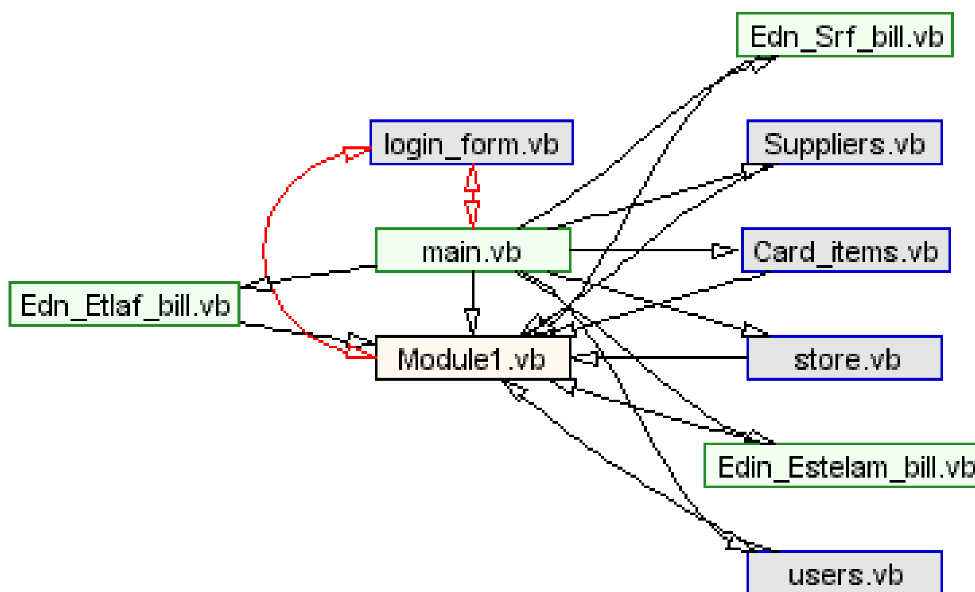


Figure 4.25: the appropriate intermediate representation of the case study using the Project Analyst application

The above figure shows the classes and forms of the system; and also, the relationships between them in terms of dependency, ie the dependence of each form on other forms. Moreover, the shares show the relationships between these classes and forms, where the black shares show the directed association relationship between classes or forms, while the red shares show the generalization (inheritance/extend) relationship between classes or forms. After that; A description in full detail of the classes and forms; And also, all the methods which are included in these classes in the following figure:

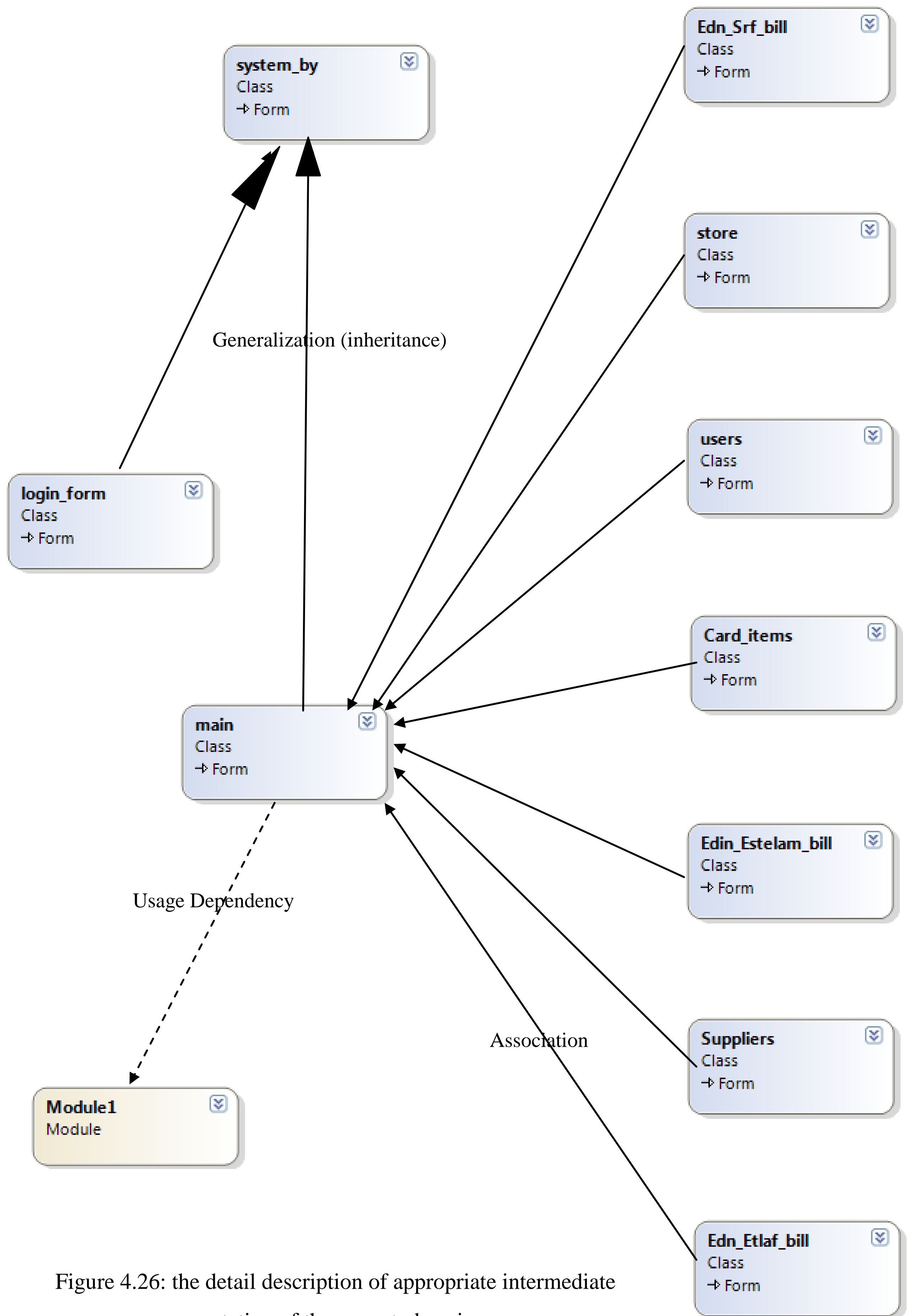


Figure 4.26: the detail description of appropriate intermediate representation of the case study using

Second Stage: **Code Restructuring Stage**

Now, the restructuring stage of the software system begins. Restructuring must be done systematically to reduce the many risks. This is done by arranging the steps that are Implemented at this stage. Which starts from detecting the smell of the code based on the situations that are proposed in this approach. And ends with the implementation of a one of proposed restructuring solutions to eliminate this smell.

Phase 1 **Detection of Code Smells**

Due to the researcher's knowledge and his deep understanding of the system used as a case study; because of that, this system is considered as one of the systems that are easy to understand for him. Therefore, The researcher does not need to use the detection technique of smells to pick up the smells that exists in codes. Thus, **the researcher relies on the consideration and observation to detect the smells in the source code** and eliminate them by applying the effective restructuring.

Phase 1. I **Potential Cases of Code Smell**

As discussed in the previous section (in phase i of the second stage), we set out as a requirement to continue that. The user is determinate the bottlenecks in source code by the consideration that actually perform the detection of code smells in software system.

The source code should be well watched, although the developer's performance might be affected (ie, it is very difficult to do this). In the beginning the system is viewed in a simple way (**The bird's look**) based on components (Classes and Forms) to determine the similar in the names of the methods in the same class or in the different classes. And also, to determine whether the method is long or not; Hence, By knowing the class methods, it is possible to determine whether the class is large or not.

Then, we turn to look more closely(**The Infrastructure Inspect**); is to look at the similarity in the source code for the method itself (Lines Of Code), if they exists

in the same class or in different classes. This is based on a set of situations that are processed by the Enhancement Approach.

- **At the beginning, the Simple Look of the System (The Bird's Look):**

The system components (Classes and Forms) in an abstract way is to be examined, and without going into details, and by looking at the system components it is found:

<u>First</u> There is a similarity in the names of the methods that are found in most forms (See the Following Figure)

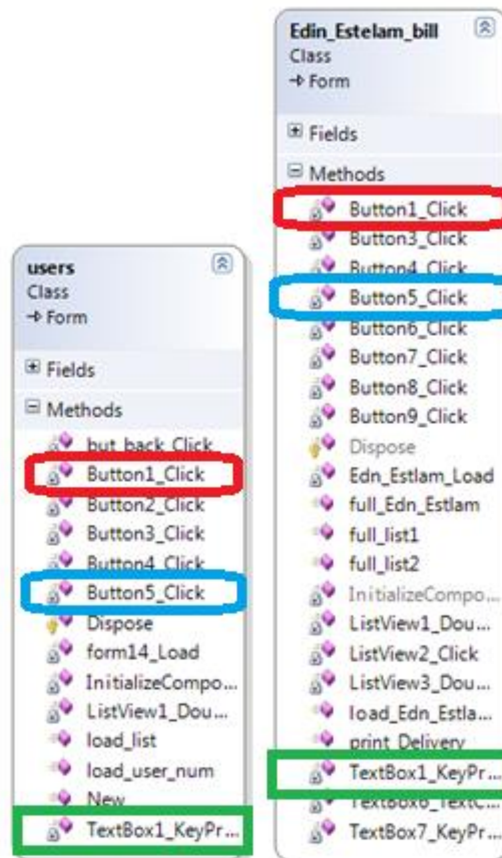


Figure 4.27: the similarity of the methods' names are in the forms

This is considered as one of the code smells which must be eliminated by using one of the improving solution (Restructuring Technique) known as Rename Method (A.1). Apply this technique in detailed in the next section (Phase 2.I).

Second There are many methods are considered as a long method(see the following figure)

```
    PrivateSub Button1_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles Button1.Click
If ListView2.Items.Count = 0 Then
            MsgBox("يجبادراجاصنافاذنالاستلام", MsgBoxStyle.Question +
MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
Exit Sub
EndIf
If TextBox3.Text = ""Then
            MsgBox("ادخلبياناتالمورد", MsgBoxStyle.Question +
MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
            TextBox3.Focus()
Exit Sub
EndIf
        load_Edn_Estlam_num()
Dim s1 AsString = "select * from Edn_Estlam where ed_estelam_no ="&
Val(TextBox2.Text)
Dim cm1 AsNewSqlCommand(s1, cn)
Dim r1 AsSqlDataReader = cm1.ExecuteReader
If r1.Read = TrueThen
            MsgBox("يرجىإعادةالحفظ", MsgBoxStyle.Information +
MsgBoxStyle.MsgBoxRight + MsgBoxStyle.MsgBoxRtlReading, "تنبيه")
            r1.Close()
Exit Sub
EndIf
        r1.Close()
Dim s2 AsString = "insert into Edn_Estlam
(ed_estelam_no,ed_Suppliers_no,ed_date,ed_user)values(@x1,@x2,@x3,@x4)"
Dim cm2 AsNewSqlCommand(s2, cn)
        cm2.Parameters.AddWithValue("@x1", Val(TextBox2.Text))
        cm2.Parameters.AddWithValue("@x2", Val(TextBox3.Text))
        cm2.Parameters.AddWithValue("@x3", (Format(DateTimePicker1.Value,
"yyyy/MM/dd")))
        cm2.Parameters.AddWithValue("@x4", user_name)
        cm2.ExecuteNonQuery()
For i = 0 To ListView2.Items.Count - 1
Dim s3 AsString = "insert into details_Estelam
(db_estelam_no,db_item_no,db_store_no,db_quantity)values(@x1,@x2,@x3,@x4)"
Dim cm3 AsNewSqlCommand(s3, cn)
            cm3.Parameters.AddWithValue("@x1", Val(TextBox2.Text))
            cm3.Parameters.AddWithValue("@x2",
Val(ListView2.Items(i).SubItems(4).Text))
            cm3.Parameters.AddWithValue("@x3",
Val(ListView2.Items(i).SubItems(5).Text))
            cm3.Parameters.AddWithValue("@x4",
Val(ListView2.Items(i).SubItems(3).Text))
            cm3.ExecuteNonQuery()
Next
For i = 0 To ListView2.Items.Count - 1
Dim qn AsInteger = 0
Dim FOUND AsBoolean = False
```

```
Dim s AsString = "select * from Items_stock where it_item_no ="&
Val(ListView2.Items(i).SubItems(4).Text) &" and it_store_no ="&
Val(ListView2.Items(i).SubItems(5).Text)
Dim cm AsNewSqlCommand(s, cn)
Dim r AsSqlDataReader = cm.ExecuteReader
If r.Read = TrueThen
                FOUND = True
                qn = Val(r!it_quantity)
                r.Close()
EndIf
            r.Close()
If FOUND = FalseThen
Dim s5 AsString = "insert into Items_stock
(it_item_no,it_store_no,it_quantity)values(@x1,@x2,@x3)"
Dim cm5 AsNewSqlCommand(s5, cn)
                cm5.Parameters.AddWithValue("@x1",
Val(ListView2.Items(i).SubItems(4).Text))
                cm5.Parameters.AddWithValue("@x2",
Val(ListView2.Items(i).SubItems(5).Text))
                cm5.Parameters.AddWithValue("@x3",
Val(ListView2.Items(i).SubItems(3).Text))
                cm5.ExecuteNonQuery()
Else
Dim s6 AsString = "update Items_stock set it_quantity=@x1 where it_item_no ="&
Val(ListView2.Items(i).SubItems(4).Text) &" and it_store_no ="&
Val(ListView2.Items(i).SubItems(5).Text)
Dim cm6 AsNewSqlCommand(s6, cn)
                cm6.Parameters.AddWithValue("@x1",
Val(ListView2.Items(i).SubItems(3).Text) + qn)
                cm6.ExecuteNonQuery()
EndIf
Next
        MsgBox("تمتعمليةالحفظ", MsgBoxStyle.MsgBoxRight, "تأكيد")
Dim da1 AsNewSqlDataAdapter("select * from Store", cn)
Dim das1 AsNewDataSet
        das1.Clear()
        da1.Fill(das1, "Store")
        ComboBox1.DataSource = das1
        ComboBox1.ValueMember = "Store.st_no"
        ComboBox1.DisplayMember = "Store.st_name"
        TextBox2.Clear()
        TextBox3.Clear()
        TextBox1.Clear()
        TextBox4.Clear()
        TextBox10.Clear()
        TextBox5.Clear()
        TextBox6.Clear()
        TextBox9.Clear()
        TextBox7.Clear()
        Label13.Text = ""
        ListView1.Items.Clear()
        ListView2.Items.Clear()
        ComboBox1.SelectedIndex = 0
        TextBox1.Enabled = True
        DateTimePicker1.Enabled = True
        DateTimePicker1.Value = Now.Date
        Button1.Enabled = True
        Button3.Enabled = False
        GroupBox3.Enabled = True
Dim da1 AsNewSqlDataAdapter("select * from Store", cn)
Dim das1 AsNewDataSet
        das1.Clear()
```

```
        da1.Fill(das1, "Store")
        ComboBox1.DataSource = das1
        ComboBox1.ValueMember = "Store.st_no"
        ComboBox1.DisplayMember = "Store.st_name"
        load_Edn_Estlam_num()
        full_list1()
        full_list2()
        full_Edn_Estlam()
    End Sub
```

Figure 4.28: the method that is considered as a long methods

There are 141 line of code in this method, these are considered as one of the code smells known as the Long Method, which must be eliminated by using one of improving solutions (Restructuring Technique) like Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object or Replace Method with Method Object. one of these techniques to be implemented in the next section (see section Phase 2). But, in this system no long method can be solved, because the source code of this method is not disassembled.

Third There are many classes considered as a large class(see the following figure)

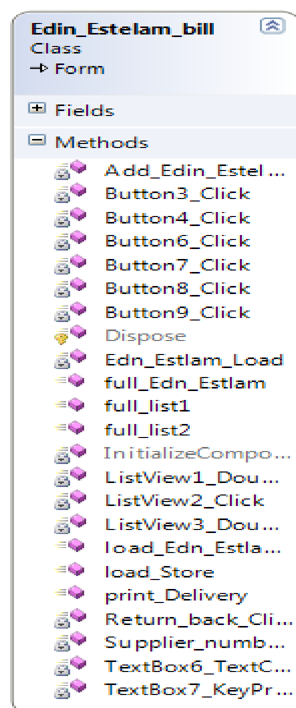


Figure 4.29: the class is considered a large class

In this class there are 23 methods. These are considered as one of the code smells known as Large Class, which must be eliminated by using one of improving solutions (Restructuring Technique) like Extract Class, Extract Subclass, Extract Interface. one of the techniques is to be implemented in the next section (see section Phase 2). But ,in this system no large class can be dismantled.

- **At the End, the Close View of the System (The Infrastructure Inspect):**

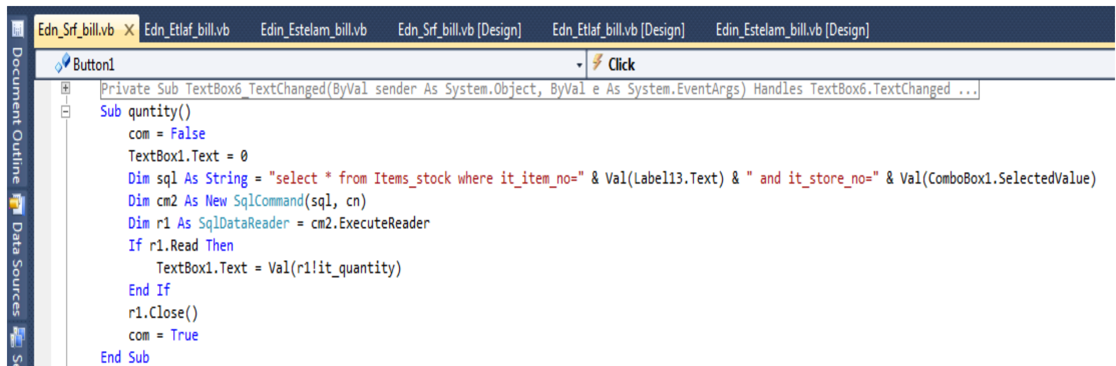
Consequently, the system components is to be viewed closely (Classes and Forms) in a detailed way. this is done by focusing the look at the code lines, Which is established by each method in the form or class, and is to look at the similarity in the source code for themethod itself (Lines Of Code). if they exists in the same class or in different classes, and by looking at the code lines are found:

First There is a similarity in the lines code that are found in different methods ,but in the same form (see the following figure)

Figure 4.30: the similarity of the code lines that are in the different methods but in the same form

This is considered as one of the code smells which must be eliminated by using one of the improving solution (Restructuring Technique) known as Extract Method (A.2). Apply this technique in details in the next section (Phase 2.II).

Second There is similarity in some of the methods that are found in different form (see the following figure)



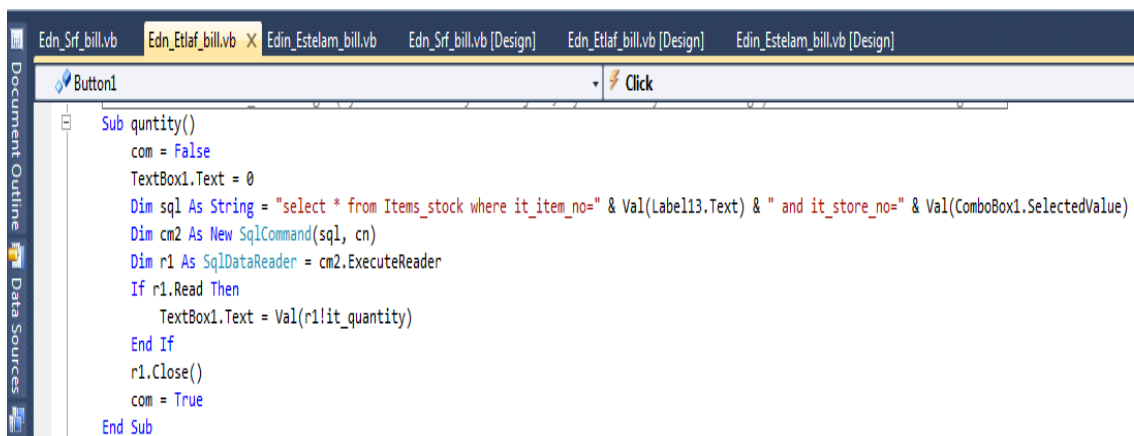


Figure 4.31: the similarity of the some methods that are found in the different forms

This is considered as one of the code smells which must be eliminated by using one of the improving solution (Restructuring Technique) known as Extract Method (A.2). Apply this technique in details in the next section (Phase 2.III).

Phase 2 **Enhancement Mechanism**

Phase 2. I **Rename Method**

1. **Bad smell**: the duplication method name.
2. **Method Name**: Rename Method.
3. **Location**: Edn_Srf_bill : Class and Edn_Etlaf_bill : Class.
4. **Reasons**: Increase the understandability and readability.
5. **Description**: It is used after an extraction in order to name the newly extracted method. Methods should be named in a way that communicates (announces) their intention (function).
6. **Restructuring Process**:

1 Find a name for the new method you extract that reflects the function of the methods.

- In this case the methods which extract in two forms:
  - ✓ **Form: Users**
    - o Add_User_Click().
    - o NewNew_Click().
    - o User_number_KeyPress().
  - ✓ **Form: Edin_Estelam_bill**
    - o Add_Edin_Estelam_bill_Click().
    - o Return_back_Click().
    - o Supplier_number_KeyPress().

2 The new methods signature that are extracted are not implemented by a super-class or sub-class. In other words, the names of these methods extracted which are not used in all software source code.

3 Declare a new method with the new name. In this case the methods which extract in two forms:

✓ **Form: Users**

o Private Sub Add_User_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Add_User.Click.

o Private Sub NewNew_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles NewNew.Click.

- o Private Sub User_number_KeyPress(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles User_number.KeyPress.

✓ **Form: Edin_Estelam_bill**

- o Private Sub Add_Edin_Estelam_bill_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Add_Edin_Estelam_bill.Click.
- o Private Sub Return_back_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Return_back.Click.
- o Private Sub Supplier_number_KeyPress(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles Supplier_number.KeyPress.

4 Moreover, copy the old body of code over to the new name. And also, there are some changes that must be made in the two forms. These changes are:The textbox1 is used by some methods, which has been changed to User_number in the Users form and to Supplier_number in Edin_Estelam_bill form; Therefore, must be use the new names in all Methods that used of textbox1 .

5 Compile

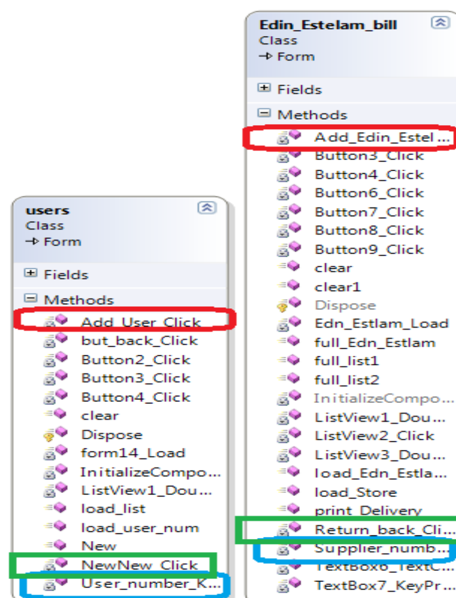


Figure 4.32: the apply of Rename Method mechanism

At the end, we will apply this mechanism in the all software system forms for resolve similar methods names problem.

Phase 2. II **Extract Method**

1. **Bad Smell**: the duplication in the Same Class.
2. **Method Name**: Extract Method.
3. **Location**: store : Class.
4. **Reasons**: Increase the performance and Maintain the consumption of computer resources.
5. **Description**: Extract Method is considered the duplication of method that must be removal; If a code fragment can be grouped together, turn it into a new method whose name explains the purpose of the this method and replace the fragment with a call to the new method.
6. **Restructuring Process**:

1. Create a new method in the same class, and name it (Methods should be named in a way that communicates their intention); In this case the lines of code that should be extracted are perform the cleans process TextBoxs that existed in the store form; Therefore, the name of new method is clear().

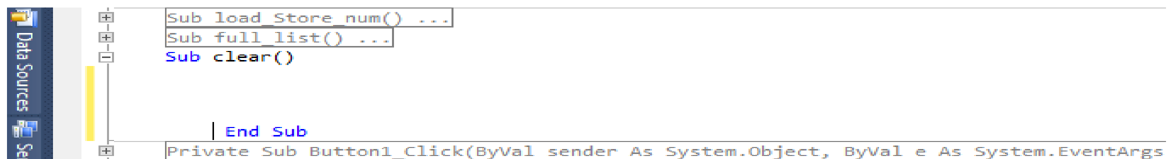


Figure 4.33: the Extract Method mechanism: Create a new method in the same class [i.e. clear()]

2. Copy the extracted code from the source method into the new target method.

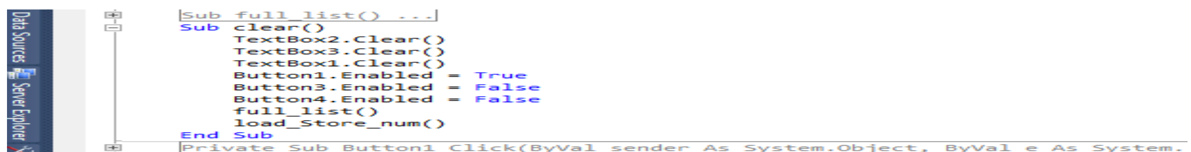


Figure 4.34: the Extract Method mechanism: Copy the extracted code from the source method into the new method

3. Scan the extracted code for references. If there are local variables will be sent as parameters of the new method, in this case there is not any local variables.
4. In this case there is not any temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. ,In this case there is not any variables that are used by new method.
6. Replace the extracted code in the source method with a call to the new method.

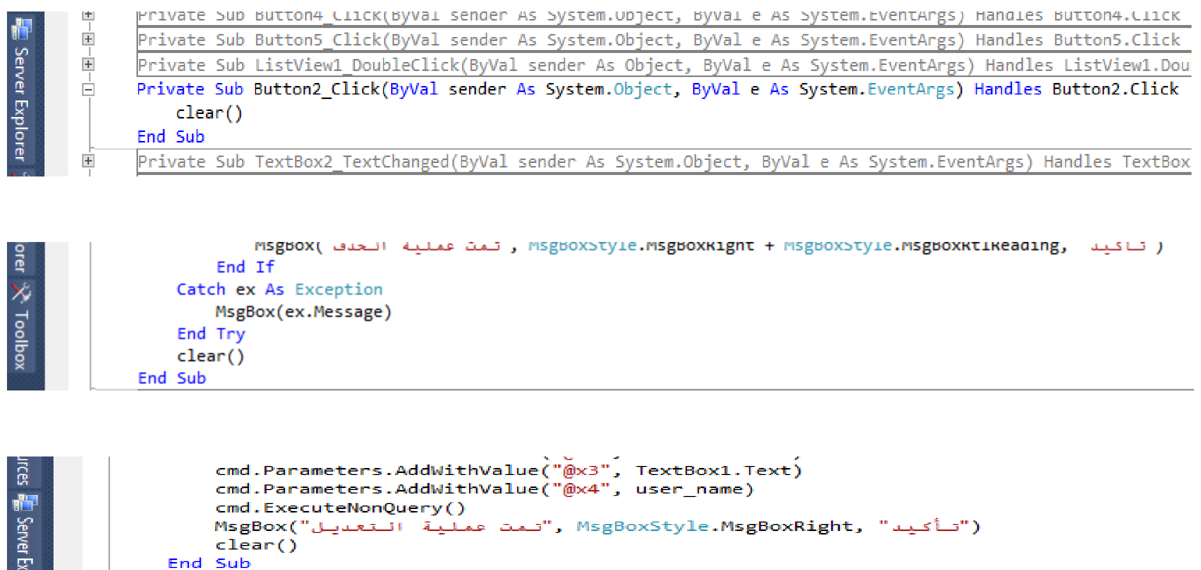


Figure 4.35: the apply of Extract Method mechanism

7. Compile and test.

At the end, we will apply this mechanism in the all software system forms for resolve similar methods names problem.

Phase 2. III **Extract Method**

1.1 **Bad smell**: the duplication in the Same Class

1.2 **Method Name**: Extract Method.

1.3 **Location**: store: Class.

1.4 **Reasons**: Increase the performance and Maintain the consumption of computer resources.

1.5 **Description**: Extract Method is considered the duplication of method that must be removal; If a code fragment can be grouped together, turn it into

a new method whose name explains the purpose of the this method and replace the fragment with a call to the new method.

1.6 **Restructuring Process**:

1. Create a new method in the super class, and name it (Methods should be named in a way that communicates their intention); In this case the name of new method is like the methods name contained the similar code lines. Therefore, the name of new method is quntity1().

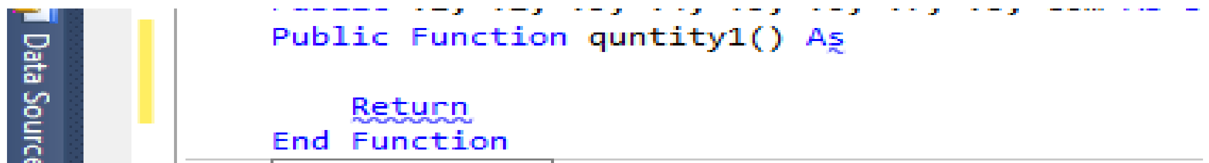


Figure 4.36: the Extract Method mechanism: Create a new method in the same class [i.e. quntity1()   ]

2. Copy the extracted code from the source method into the new target method.

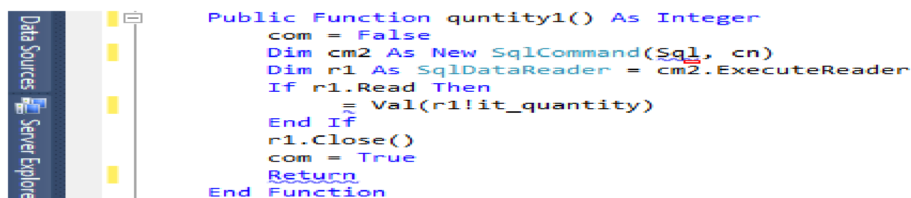


Figure 4.37: the Extract Method mechanism: Copy the extracted code from the source method into the new method

3. There is a local variable its name is sql, sent it as parameters of the new method.

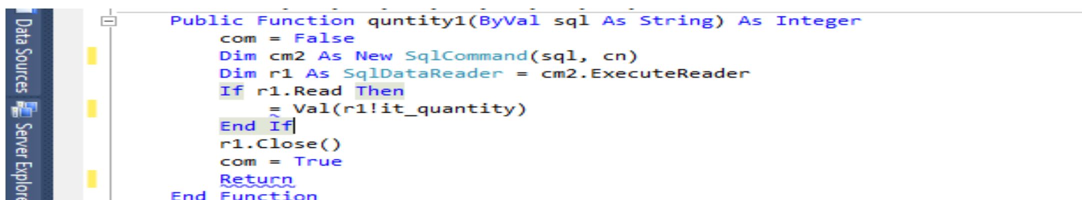


Figure 4.38: the Extract Method mechanism: send local variable as parameters to the new method

4. There is a temporary variable it name is com, declaret it in the target method as temporary variables.
5. Now, the new method must be declared the new integer variable (I) for return back Holds the result of this method.

```
Public Function quntity1(ByVal sql As String) As Integer
    com = False
    Dim I As Integer
    Dim cm2 As New SqlCommand(sql, cn)
    Dim r1 As SqlDataReader = cm2.ExecuteReader
    If r1.Read Then
        I = Val(r1!it_quantity)
    End If
    r1.Close()
    com = True
    Return I
End Function
```

Figure 4.39: the Extract Method mechanism: define the new integer variable for return back Holds the result of this method

6. Replace the extracted code in the source method with a call to the new method.

```
Edn_Srf_bill.vb   Edn_Etlaf_bill.vb X   Module1.vb
Edn_Etlaf_bill                          quntity
        Dim sql As String = "select * from Items_stock where it_item_no=" & Val(Label13.Text) & " and it_store_no=" & Val(ComboBox1.SelectedVal
        Me.TextBox1.Text = quntity1(sql)
    End Sub
```

```
Edn_Srf_bill.vb X   Edn_Etlaf_bill.vb   Module1.vb
Edn_Srf_bill                            quntity
        Dim sql As String = "select * from Items_stock where it_item_no=" & Val(Label13.Text) & " and it_store_no=" & Val(ComboBox1.SelectedVal
        Me.TextBox1.Text = quntity1(sql)
    End Sub
```

Figure 4.40: the apply of Extract Method mechanism

7. Compile and test.

At the end, we will apply this mechanism in the all software system forms for resolve similar methods names problem.

Phase 3 **Application Solution**

Now, after finishing removing the smells that appeared in the source code. There are two important things to be returned:

Firstly: in all source codes, the code lines that are used to link to the data baseis be returned (See Figure 4.41)

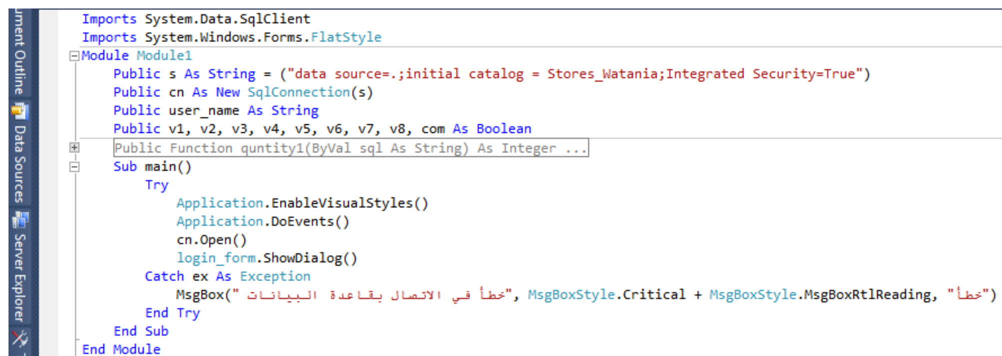


Figure 4.41: return the code that is used to link of the database

Note:

✓ This technique (mechanism) is to be applied to all the codes in the target software system.

Secondly: some important developer comments that have been omitted from the source code are returned (See the following Figure).

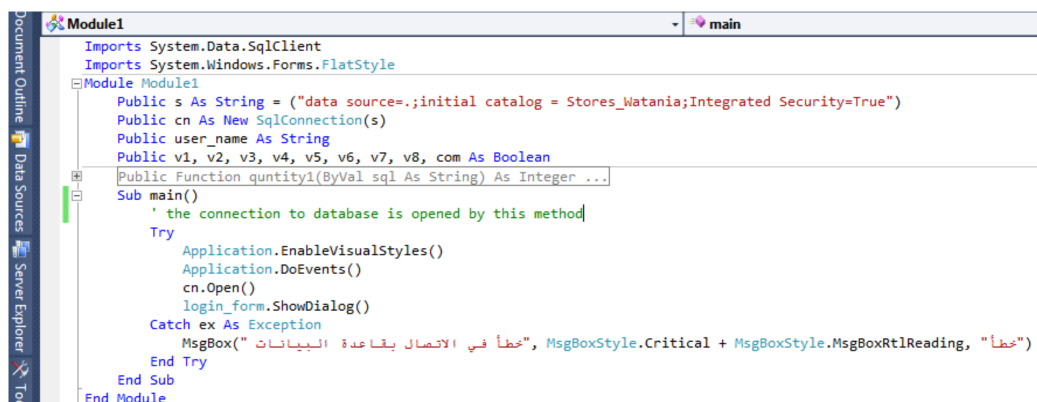


Figure 4.42: return some important developer comments that have been omitted

CHAPTER 5

The Quantitative Validation of the Enhancement Approach

5.1 Introduction

This chapter discusses the ***Quantitative Validation*** of proposed Approach using object oriented metrics (source code calculation metrics). The whole process is based on the size of program code and number of files that created the system, these measured by some of object-oriented metric criteria like Line Of Code (LOC) metrics, Blank lines, Executable Physical, Executable Logical and McCabe VG Complexity (these metrics known as Internal Measures), that are helping in verification Enhancement Approach based on metrics.

There are several purposes in attempting to evaluate Enhancement Approach:

- ✓ To determine what the advantages and disadvantages of the Approach.
- ✓ It also help researchers to verification that the proposed Approach is effective or ineffective.

The object-oriented metric criteria, therefore, are to be used to answer the research questions for evaluating the software code.

5.2 Presentation of the Results:

For the presentation of the results, the effective tools to measure the cases study are used that, by verification in the previous chapter; these Tools are:

1. *Project Analyzer Tool (version10.2).*
2. *Ndepend Tool.*

❖ Project Analyzer Tool

" Project Analyzer is a Visual Basic code review and quality control tool. Understand, optimize and document your Visual Basic code. Project Analyzer reads source code written with Visual Basic versions 3.0–6.0 and VB.NET 2002–2013. Office VBA is supported with VBA Plug". ***(Service Manual)***

✓ What are the main benefits?

Project Analyzer makes a full code review. Project Analyzer generates technical documentation by reading program source code. The available documents include graphical representations of program structure, commented source code listings and various reports such as file dependencies. Automatic document generation relieves the programmers from the burden of keeping technical documentation in sync with the existing code. [85]

Project Analyzer helps programmers to understand existing code in less time. By browsing code in hypertext form and viewing interactive graphs, a programmer can quickly understand how a certain function operates with other functions and variables. ***This helps evaluate the impact of code changes***. It is also useful for understanding the migration effort from classic VB to VB.NET. [61][85]

✓ Project Metrics

To monitor their programming efforts, software engineers often use some simple metrics such as lines of code or EXE size. These are the most basic metrics. They aren't very sophisticated, but they're easy. Project Analyzer knows more. It can tell you about the understandability, complexity and reusability of your code. [61]

" Project Metrics provides more than a hundred different metrics. You can find comprehensive instructions to metrics and their use in the help file in the Tool. Some of the available metrics are:" ***(Service Manual)***

- **Size metrics**, such as lines of code and number of methods.

- **Complexity metrics**, such as McCabe cyclomatic complexity, cyclomatic density, depth of conditional nesting, structural fan-in/fan-out, informational complexity, class hierarchy metrics.
- **Understandability metrics**, such as length of names and amount of comments.

Therefore, it is relied on this tool to give the validation results of the case study before and after apply of the Enhancement Approach. The following section present the Comparison report that is presented by this tool.

5.3 Project Status Report

Before Using Enhancement Approach <2017-10-11> vs. [After Using Enhancement Approach <2017-10-13>] :

5.3.1 System Size

There are 4,863 *[2,866]* lines in the system (LLINES). It is a small system. Of these lines, 3,328 *[2,801]* are code, 458 *[57]* are pure comment lines and 1,077 *[8]*are empty. Thus, 68% *[98%]* is code lines, 9% *[1.99%]* is comment lines and 22% *[0.28%]* is empty lines.

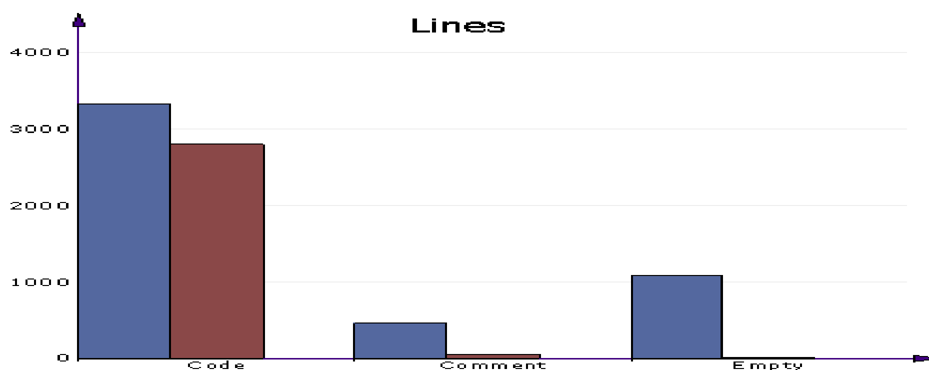


Figure 5.43: Charts are illustrating the distribution of the system size before and after the implementation of the Enhancement Approach

Other size metrics:

- ✓ Number of statements: 3,338 *[2,811]* (STMT).
- ✓ Number of procedures: 139 *[148]* (PROCS).
- ✓ Kilobytes of source code: 182 *[140]* kB.
- ✓ Source files: 89 *[90]*.
- ✓ .Forms: 6 *[6]*.

5.3.2 Commentation

All in all, there are 248 *[56]* meaningful comments (MCOMM) in the system (compared to 3,328 *[2,801]* code lines). By meaningful we mean a comment that has some text in it, not just separators or empty comments.

The comment density (meaningful comments per code line, MCOMM%) is 7% *[2.00%]*. In other words, there is a comment for every 13.4 *[50.0]* lines of code. We recommend a density of over 20% so that there is at least 1 comment for every 5 lines of code.

Anything to improve? There are 10 *[10]* files with comment density less than 15%. You should consider adding more cementation to files that fall below this limit. There are 33 *[130]* procedures having no cementation.

5.3.3 Complexity

The average cyclomatic complexity (CC) of a procedure is 2.8 *[2.6]*.

Anything to improve? There are 2 *[2]* moderate risk procedures (CC 11..20), no *[0]* high risk procedures (CC 21..50) and no *[0]* very high risk procedures (CC over 50). The total number of procedures is 139 *[148]*.

Cyclomatic complexity (CC) is counted as decisions+1. Decisions include statements such as If, ElseIf, Case, For, While and Until. The higher CC, the riskier are code changes to that procedure. If CC exceeds 20, you should consider it

alarming. Procedures with a high cyclomatic complexity should be simplified or split into several smaller procedures.

5.3.4 **Conditional Nesting**

The average depth of conditional nesting (DCOND) is 0.9 *[0.8]*. Thus, on average, there are so many nested conditional statements (nesting levels) in a procedure.

Anything to improve? There are no *[0]* procedures with DCOND>5. Too many nesting levels make the code difficult to understand and can lead to errors in program logic. Consider splitting these procedures. You may also find a way to rewrite the logic with a Select Case statement or an easier-to-read If..Then..ElseIf..Else structure.

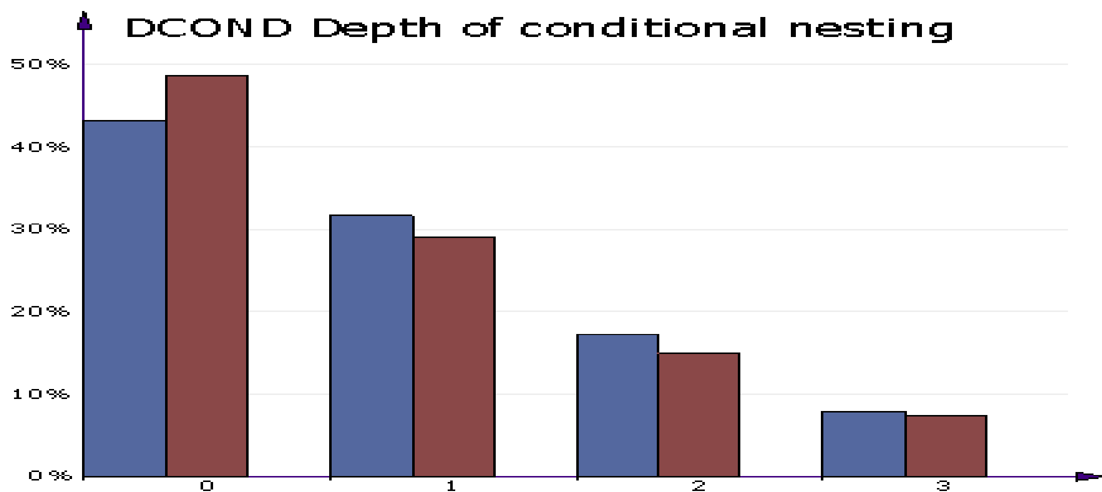


Figure 5.44: the average depth of conditional nesting (DCOND)

5.3.5 **Procedure Length**

The average procedure (LINES/proc) is 31.8 *[18.7]* lines. Shorter procedures are easier to understand than longer ones.

Anything to improve? There are 19 *[4]* procedures that exceed one page when printed (66 lines).

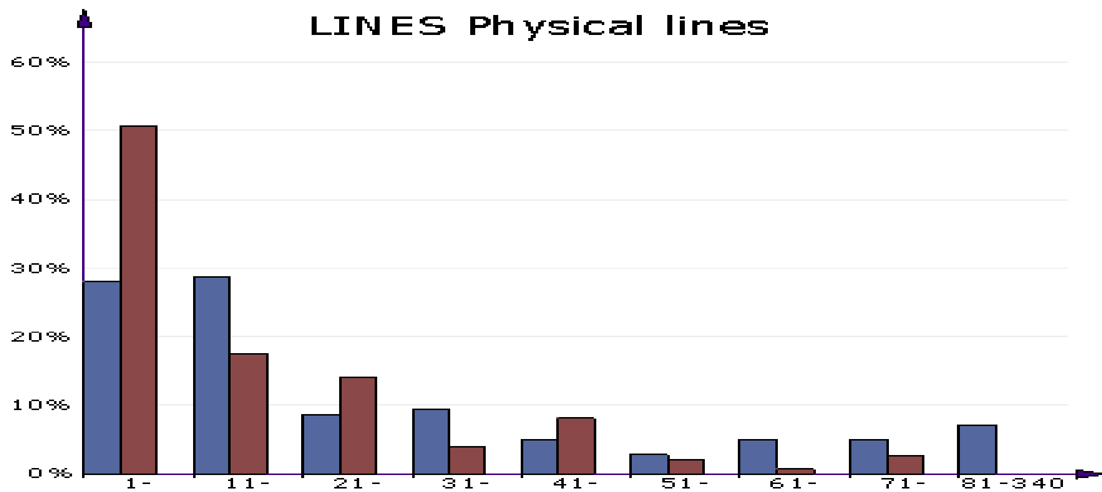


Figure 5.45: the average procedure length (LINES/proc)

5.3.6 File Length

The average file (LINES/file) is 486.3 *[286.6]* lines. Shorter files are easier to understand than longer ones.

Anything to improve? There are no *[0]* files that exceed 1000 lines. No *[0]* file has over 50 procedures. No *[0]* file declares over 50 variables and no *[0]* file declares over 50 constants.

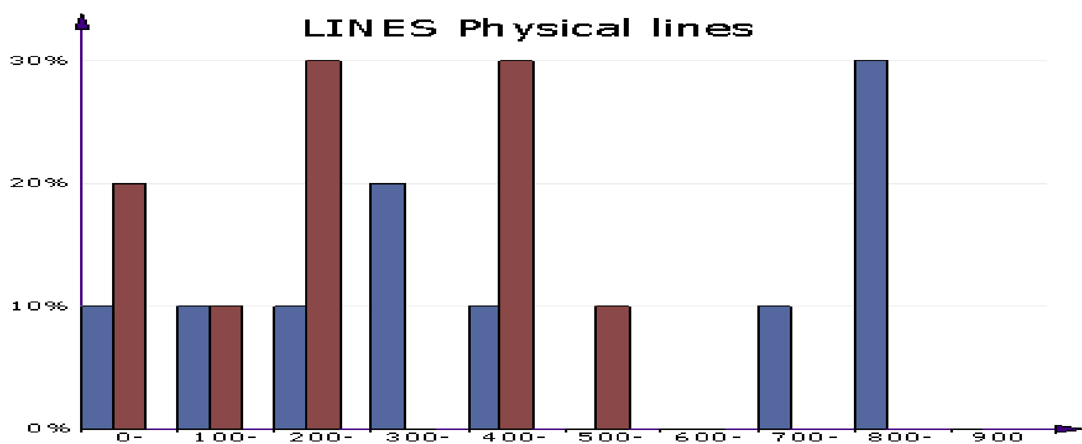


Figure 5.46: the average file length(LINES/file)

5.3.7 **Parameters**

The average number of procedure parameters (PARAMS) is 1.6 *[1.4]*.

Anything to improve? There should be max 5 parameters in a procedure. In this system, this count is exceeded in no *[0]* procedures. Consider simplifying or splitting those procedures.

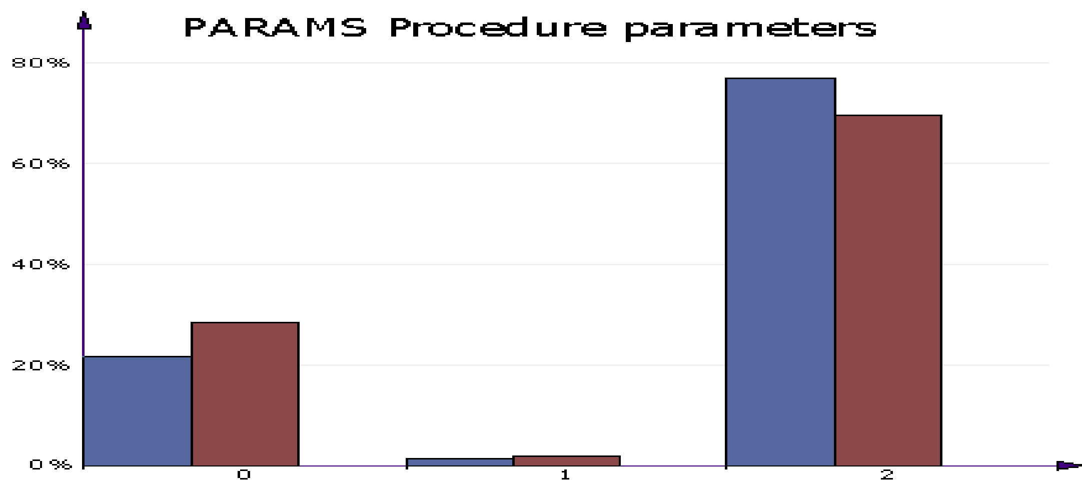


Figure 5.47: the average number of procedure parameters (PARAMS)

5.3.8 **Class Design**

The maximum depth of class inheritance tree (maxDIT) is 7 *[7]*. This should be 6 or fewer.

Class variables should always be declared private to avoid accidental changes. In this system this succeeds at a rate of 46% *[44%]*. Of all the 9 *[9]* classes, 2 *[2]* classes have problems in this regard.

Project Metrics Viewer v4.1.05

Now, Ndepend Tool will be used to compare the coupling and complexity of the study case before and after the Enhancement Approach is applied :

5.3.9 **Coupling Metrics:**

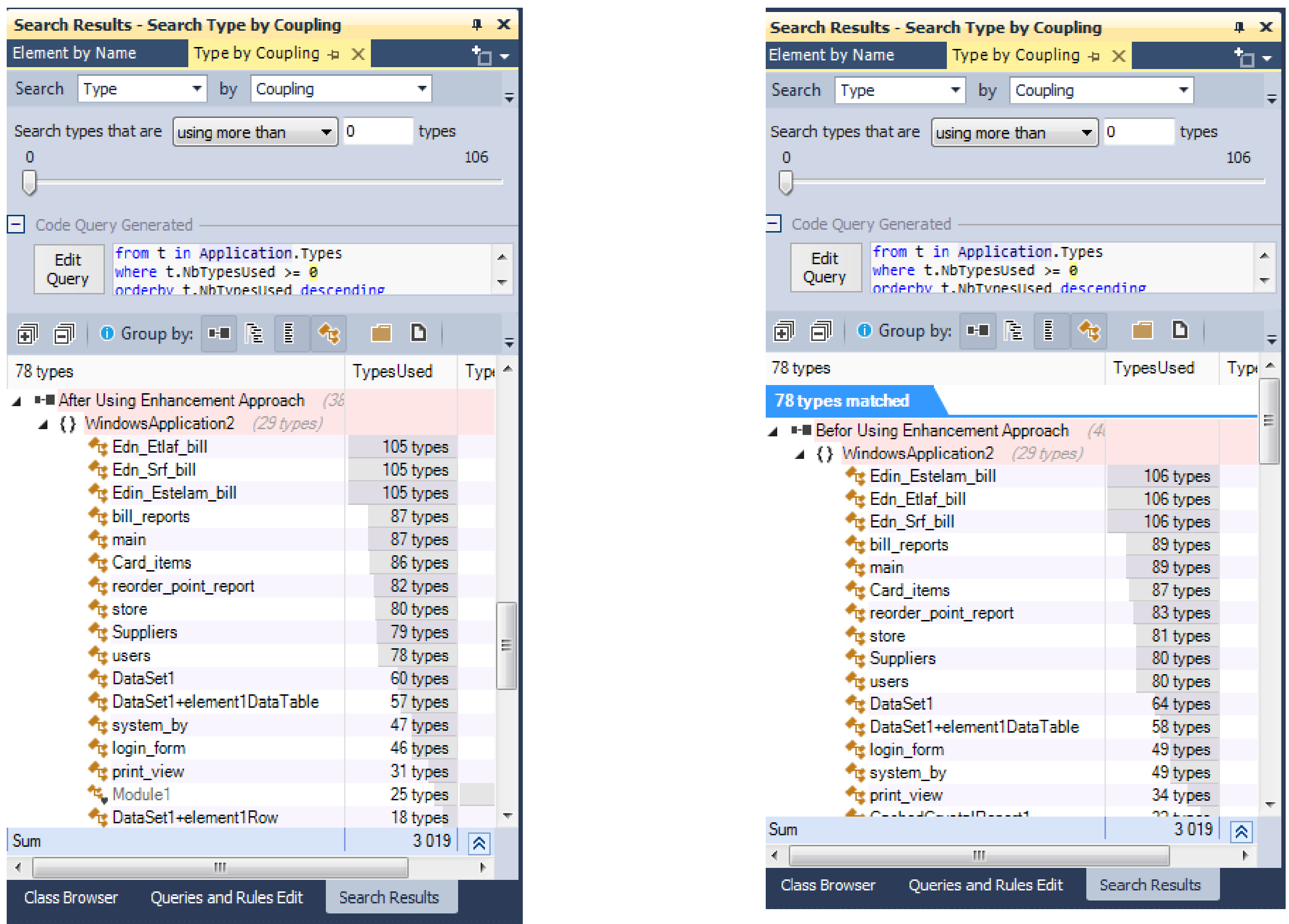


Figure 5.48: the coupling metrics

CHAPTER 6
Conclusions and Future work

6.1 Aanalysis

In order to answer the research questions, a case study has been conducted for the general Mill company (chapter 4) the proposed approach has been used to improve the system, which allowed for the opportunity of evaluation by analyzing the results using the effective tools depending on the concepts of quantitative investigation for code programming (chapter 5), and by comparing analysis results for the system before and after improvement approach. What has been mention above enable us to answers the research questions.

RQ 1: Size- Does the existence of the code smells make the source code large? And; Does the restructuring of the source code make it smaller?

Point 5.3.1 shows the size of source code before and after approach application. Regarding the form of this point, and by comparing results it could be seen, firstly: the size of the source code is 182KB. However, as soon as the approach is applied and deletion of smells in the source code, the size of the source code of the organizer decrease to 140 KB this decrease is excellent , because the used system as a case study is not considered as a big system. If this approach is applied on a bigger system the percentage of decrease in the code size would be bigger as compared to the code size. Secondly: the number of program lines LOC of the system has decreased from 3338 to 2811.

RQ 2: Complexity - Is the complexity of the system affected by the size of the smells that exist in the source code?

Point 5.3.3 shows the source code of the system’s complexity before and after applying the improvement approach. As in regard to the calculated results using the CC , and taking their the mean average and analyzing them, it could be seen: the complexity mean average of the system is 2.8, however, as soon as applying the improvement approach in the complexity mean average it will decrease to 2.6.

This indicates that the co-relation (If A increases, B increases) between the size of the smells and complexity is a reverse co-relation (as soon as the size of the smells increases it becomes more complexity) in other words, complexity is effected by the size of the smells.

RQ 3: Software Reliability-Does the program work without failure after applying the suggested restructurings on the program? (Probability of failure-free operation of a computer program for a specified time in a specified environment).

After the use of Enhancement Approach, which executes the restructurings concepts, and by applying it to the general Mills Company, this system which is has used, and testing the functions it performs, it has been found out that the system is functioning without failure. This indicates that the restructuring application on the programming systems does not impact the external functions behaviour this is part of the definition, in addition, that it supports software reliability.

RQ 4: Maintainability - How good are code smells as indicators of system-level Maintainability of software?

As a matter of fact, maintainability is considered a unit of quality measurement "qualitative Validation" of the source code which has been left out for future research. However, all the researchers, developers and those concerned with system designs and constructions keep in mind maintainability.

Actually the probability of maintainability means the simplicity of maintaining defaults and how to develop the program in the future, as well as, the probability of maintainability is effected by the complexity of the system, the more the system is complex the harder its maintained. i.e. the probability of maintainability is co-related with complexity. As in respect to the item 5.3.3 its found that the rate of the system complexity before the application of the suggested approach was 2.8 and became 2.6 after applying the approach. As mentioned before (adverse relation (If A increaes B decreases and vica versus) between complexity and maintainability) this indicates that the reduction of complexity percentage after applying the suggested approach means that maintainability has increased and improved.

6.2 Conclusion

Smell code is one of the elements that acutely complicates the maintainability and development of programming systems, it is also considered one of the structuring in programming as indicated to as principles of basic designs, which effects negatively the design quality. In this research, a simple approach has been provided which depends on definitions of reverse engineering, and restructuring to analyze and eliminate smells in the source code, through defining a set of situations for that smell. The situation is defined as the relation between classes that contain smells.

The proposed approach is considered a hybrid approach, it includes two sort of approaches or techniques (Graph and text-based approach). Graphs are used to describe all situations to facilitate smells detection, because it uses texts in describing effective restructuring for all situations (a situation represent a specific case of code smell). The developed General Mill's Company system has been analyzed by using VB.Net 2010 and by adding the suitable improvements through our approach, also the qualitative validation has been ratified this system before and after improvement.

The actual result for the experiment on the case study, shows that using the situation provides useful information which helps in detecting code smell; Also the use of reverse engineering is considered effective, because it support detection of smells a siple visual way. Finally, definitions of restructuring use in a textually makes it simple to eliminate smells effectively. In addition to using (point 2.3 page 57) makes smells elimination of these smells is done in an organized way, because it describes every smell detail and how to eliminate it. In addition to the fact that the test and verifications of results quantitatively provided good results that presented all answers to this research. Therefore, this research has achieved its objectives represented in improving an approach that would help developers in improving their source code of the system through eliminating three source of code smells (the repeated code, the long indicative, the big class) by depending on the definitions of reverse engineering, and the definition of code restructuring.

6.3 Future Work

This section summarizes ideas for potential future work. This list does not include minor improvements or cosmetic changes that are in the 'to do' list for Enhancement Approach. Also omitted are various planned internal design changes (e.g. to improve maintainability or efficiency of the system). Future work of this research:

1. *Further develop the Enhancement Approach for other code smells, because the any smell is appear in the code has a negative impact on the quality system standards.*
2. *The addition of other UML diagrams for use in the Enhancement Approach.*
3. *Work on integrating the Enhancement Approach into the software development life cycle (SDLC) for eliminate code smells in the software system from the beginning of SDlC, precisely at the writing code phase (Implementation Phase).*
4. *Further develop the Enhancement Approach for multi-agent systems, because the complexity in the multi-agent systems differs from other systems.*
5. *Developing a tool as a computerized software and demonstrate how to make it available as a useful technology instrument for a wide range of developers, to do the same jobs Enhancement Approach.*
6. *The addition of the quality assurance for the Enhancement Approach.*
7. *The Qualitative Validation of the Enhancement Approach will be keep in mind in the future.*
8. *Decrease risk factor by the Enhancement Approach.*

Bibliography

[1]Kaur,J and Singh,S. (2016). "Neural Network based Refactoring Area Identification in Software System with Object Oriented Metrics". Indian Journal of Science and Technology. Vol 9\1.

[2] Suryanarayana,G and et al. (2011). "Towards a Principle-based Classification of Structural Design Smells". Journal of Object Technology. Vol. 12\2.

[3]Tsantalis,N and et al. (2009)."Identification of Move Method Refactoring Opportunities". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,. Vol. 35\3.

[4]Wani,S,N and Dang,S. (2015). "A Comparative Study of Clone Detection Tools ". International Journal of Advance Research in Computer Science and Management Studies. Vol. 3\1.

[5]Al-Sreenu,K and Jagannadha Rao,D. (2012)." Performance - Detection of Bad Smells In Code for Refactoring Methods ". International Journal of Modern Engineering Research (IJMER). Vol. 2\5.

[6]Patil,V and et al. (2014). "Code Clone Detection using Decentralized Architecture and Parallel Processing-Latest Short Review". International Journal of Advanced Research in Computer Science and Software Engineering. Vol. 4\9.

[7]Al-Najdawi,N and et al. (2016). "A Frequency Based Hierarchical Fast Search Block Matching Algorithm for Fast Video Communication". (IJACSA) International Journal of Advanced Computer Science and Applications. Vol. 7\4.

[8]Ashtaputre, P and et al. (June, 2016)." An Effective Approach to Find Refactoring Opportunities for Detected Code Clones ". International Journal of Innovative Research in Science, Engineering and Technology, 6/5.

[9]Jelodar,H and Aramideh,J. (2014). " Common Techniques and Tools for the Analysis of Open Source Software in Order to Detect Code Clones: A Study". International Journal of Electronics and Information Engineering. Vol. 1\2.

[10]Moha, N and et al. (January/February, 2010). "DECOR: A Method for the Specification and Detection of Code and Design Smells". US: IEEE Transactions on Software Engineering, 36/1:20-36.

[11] Kaur,B and Kaur,H. (2015). "Clone Detection in UML Sequence Diagrams Using Token Based Approach ". International Journal of Advanced Research in Computer Science and Software Engineering. Vol. 5\5..

[12]Deborah, L and et al. (November, 1997). "An Approach for Exploring Code-Improving Transformations". University of Pittsburgh: ACM Transactions on Programming Languages and Systems, 19/6:1053-1084.

[13]Rani, A and Kaur,H. (January, 2010). "Refactoring Methods and Tools ". International Journal of Advanced Research in Computer Science and Software Engineering,2/12.

[14]Singh,G and Ali,J. (2015). "A Novel Composite Approach for Software Clone Detection". International Journal of Computer Applications. Vol. 126\7.

[15]Ducasse,S and et al. (2006).On the effectiveness of clone detection by string matching. JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTIC. 18:37–58.

[16]Fowler,M and et al. (1999). "Refactoring:Improving the Design of Existing Code". Mass: Addison-Wesley.

[17]Rani, A and Kaur,H. (December, 2012)." Refactoring Methods and Tools ". International Journal of Innovative Research in Science, Engineering and Technology, 2/12.

[18]Ten Step. "Implement Phase". http://www.lifecyclestep.com/open/450.0IMPLEMENTPHASE.htm. viewed 8/10/2016.

[19]Sharma,A and et al. (2010). " A Complexity measure based on Requirement Engineering Document". JOURNAL OF COMPUTER SCIENCE AND ENGINEERING. Vol. 1\1.

[20]Niu,Z and et al. (2012). "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. Vol. 38\1.

[21]Wikipedia the Free Encyclopaedia. (August, 2014)" Software Development". https://en.wikipedia.org/wiki/Software_development. viewed 8/10/2016.

[22]Astuti, H and et al. (March, 2015). " SOFTWARE QUALITY MEASUREMENT AND IMPROVEMENT USING REFACTORING AND SQUARE METRIC METHODS ".Journal of Theoretical and Applied Information Technology, 37/5.

[23]Adhikary,C and et al. (2014). "Detection of Clones in Digital Images". International Journal of Computer Science and Business Informatics. Vol. 9\1.

[24]Kaur, H and Kaur,P. (May, 2014). "A Study on Detection of Anti-Patterns in Object-Oriented Systems ". International Journal of Computer Applications, 93/5.

[25]Ragunath,P and et al. (2010). Evolving A New Model (SDLC Model-2010) For Software Development Life Cycle (SDLC). IJCSNS International Journal of Computer Science and Network Security. VOL.10 No.1.

[26]Fowler,M and Scott ,K and et al. (2000). "UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language. Reading ". Mass: Addison-Wesley.

[27]Stocker,M. (August, 2015). " Refactoring for Software Design Smells Review and Q&A with the Authors". https://www.infoq.com/articles/refactoring-for-design-smells-book-review viewed 8/10/2016.

[28] Ghosh,A and Lee,Y. (2017). "An Empirical Study of a Hybrid Code Clone Detection Approach on Java Byte Code ". GSTF Journal on Computing (JoC). Vol. 5\2.

[29]Zanoni,M and et al. (2012). "Automatic detection of bad smells in code: An experimental assessment". Journal of Object Technology. Vol. 11\2.

[30]Navita,M. (2017). "A Study on Software Development Life Cycle & its Model". International Journal of Engineering Research in Computer Science and Engineering (IJERCSE). Vol 4/9.

[31]McConnell,S. (2004) . In Engelman.L and Van Steenburgh.R(eds). "Code Complete--2nd ed".Redmond, Washington: A Division of Microsoft Corporation,Microsoft Press.

[32]Kannangara,S and Wijayanake,M. (2015). " AN EMPIRICAL EVALUATION OF IMPACT OF REFACTORING ON INTERNAL AND EXTERNAL MEASURES OF CODE QUALITY". International Journal of Software Engineering & Applications (IJSEA). Vol. 6\1.

[33]Baumann,C. (November, 2012) ."Framework for Automated Code Smell Correction in a BrownVeld Context".Magdeburg: Institut für Technische und Betriebliche Informationssysteme (ITI), Guericke University.

[34]AlHakami,H and et al. (2014). " AN EXTENDED STABLE MARRIAGE PROBLEM ALGORITHM FOR CLONE DETECTION". International Journal of Software Engineering & Applications (IJSEA). Vol. 5\4.

[35]Office of Information Services. (March, 2008) " SELECTING A DEVELOPMENT APPROACH " . centers for medicare and medicaid services.

[36]Meyer,B. "Object-Oriented Software Construction SECOND EDITION".Santa Barbara (California),USA:Interactive Software Engineering Inc. (ISE).

[37]Ouni,A and et al. (2015)." Improvingmulti-objectivecode-smellscorrectionusingdevelopmenthistory". The Journal of Systems and Software, 105:18-39

[38]Yang,Y and et al. (2009) ."Identifying Fragments to Be Extracted from Long Methods". 16th Asia-Pacific Software Engineering Conference.

[39]Davis,N. (December,2005) ."Secure Software Development Life Cycle Processes: A Technology Scouting Report ". Carnegie Mellon University: Software Engineering Process Management.

[40]Myagmar,S and et al. (2006). " CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. Vol. 32\3.

[41]Roy,C and et al. (February, 2009)."Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach ". Canada: School of Computing, Queen’s University.

[42]Fahim,M and Kumar.C. (December,2005) ."A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring". Canada: Department of Computer Science, University of Saskatchewan, Saskatoon, SK.

[43]Zibran,M and Roy.C. (2013) ."Conflict-aware optimal scheduling of prioritised code clone refactoring11th IEEE International Working Conference on Source Code Analysis and Manipulation. ISSN: 1751-8806.

[44]Rieger,Mand et al."A Language Independent Approach for Detecting Duplicated Code ". University of Berne: Software Composition Group.

[45]Khanna.V and et al. (September,2014) ."International Journal of Advanced Research in Computer Science and Software Engineering ". India: International Journal of Advanced Research in Computer Science and Software Engineering, 9/4. ISSN: 2277 128X

[46]Srikanth,N and et al. (July,2011) ."Conceptual Cohesion of Classes in Object Oriented Systems ". India: International Journal of Computer Science and Telecommunications,4/2 . ISSN: 2047-3338

[47]Neukirchen,H and Bisanz,M."Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites ". Germany: Software Engineering for Distributed Systems Group, Institute for Informatics, University of G¨ottingen.

[48]Dexun,Jand et al. (September,2013)."DETECTION AND REFACTORING OF BAD SMELL CAUSED BY LARGE SCALE ". China: International Journal of Software Engineering & Applications (IJSEA),5/7.

[49]Lassenius,Cand et al. (2004)."Bad Smells - Humans as Code Critics ". Finland: Helsinki University of Technology, the 20th IEEE International Conference on Software Maintenance (ICSM’04) 1063-6773/04.

[50]Rochimah,Sand et al. (2015)."Non-Source Code Refactoring: A Systematic Literature Review ". International Journal of Software Engineering and Its Applications, 6/9, pp. 197-214.

[51]Mika,V. (2015)."Empirical Software Evolvability – Code Smells and Human Evaluations ". Finland: School of Science and Technology, Aalto University.

[52]Smith,C and Williams,L. "SOFTWARE PERFORMANCE ENGINEERING ". The USA ,Santa Fe: Performance Engineering Services.

[53]Bellon,S and et al. (2007). "Comparison and Evaluation of Clone Detection Tools". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. Vol. 33\9.

[54]KONTOGIANNIS,K and et al. (1996)."Pattern Matching for Clone and Concept Detection ". Automated Software Engineering, 3, 77–108

[55]Bellon,Sand et al.(SEPTEMBER, 2007)"Comparison and Evaluation of Clone Detection Tools ". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 9/33.

[56]Al-Najdawi,N. (2012). "A Novel Hierarchical Search Algorithm for Video Compression". International Conference on Advances in Computer and Electrical Engineering. Vol. 17\18.

[57]MOTOGNA.S and et al.(2015)"METRICS-BASED REFACTORING STRATEGY AND IMPACT ON SOFTWARE QUALITY". STUDIA UNIV. BABES-BOLYAI, INFORMATICA, Volume LX, Number 2.

[58]Rajakumari,K and Jebarajan,T. (Mar, 2011)." Importance Of String-Based Techniques In Clone Detection ". Int. J. on Recent Trends in Engineering & Technology, 01/05.

[59]Chen,X and et al. (2014). " A Replication and Reproduction of Code Clone Detection Studies". Proceedings of the Thirty-Seventh Australasian Computer Science Conference. Vol. 147.

[60]Sun,Wand et al. "Analyzing Behavioral Refactoring of Class Models ". Colorado State University, Fort Collins, USA.

[61]Aivosto-programming Tools for Software Developers. (December, 2015). "Project Analyzer – VB code review". http://www.aivosto.com// . viewed 13/8/2017.

[62]Nguyen,Hand et al. (May,2009)."Complete and Accurate Clone Detection in Graph-based Models ". Canada: Electrical and Computer Engineering Department at Iowa State University.

[63]Falke,Rand et al. (October,2006)."Clone Detection Using Abstract Syntax Suffix Trees ". Germany: University of Bremen.

[64]Sarkar,Mand et al. (September,2013)."Reverse Engineering: An Analysis of Static Behaviors of Object Oriented Programs by Extracting UML Class Diagram ". International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970) Volume-3 Number-3 Issue-12.

[65]Kumar,R and et al. (Nov,2016)."An Empirical Study of Bad Smell in Code on Maintenance Effort ". International Journal of Computer Science Engineering (IJCSE), 6/5, ISSN : 2319-7323.

[66]Arendt,Tand et al. (January,2009)."Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study ". Germany: Mathematics and ComputerScience, Software-Engineering.

[67]Slinger,S. (March,2005)."Code Smell Detection in Eclipse ". Delft University of Technology: Faculty of Electrical Engineering, Mathematicsand Computer Science Department of Software Technology.

[68]Jubair,J and Khair Eddin,M. (March,2004)."CHIDAMBER-KEMERER (CK) AND LORENZEKIDD (LK) METRICS TO ASSESS JAVA PROGRAMS ". Jordan: King Abdullah II School for Information Technology, University of Jordan.

[69] Abd El-Aziz,Rand et al. (2012)."Clone Detection Using DIFF Algorithm For Aspect Mining ". Cairo, Egypt: (IJACSA) International Journal of Advanced Computer Science and Applications, 8/3.

[70]Sethi,D and et al. (July, 2012). " Detection of code clones using Datasets ". International Journal of Advanced Research in Computer Science and Software Engineering, 7/2 ,ISSN: 2277 128X

[71]Dobrza,L. (July, 2005)."UML Model Refactoring- Support for Maintenance of Executable UML Models ". Sweden: School of Engineering,Blekinge Institute of Technology.

[72]Kaur,H and kaur,R . (Aug, 2014)." A Review: Clone Detection in Web Application Using Clone Metrics". International Journal of Computer Science Trends and Technology, 4/2.

[73]Zibran,M and Roy,C. (April, 2013). " Conflict-aware optimal scheduling of prioritised code clone refactoring". 11th IEEE International Working:Conference on Source Code Analysis and Manipulation ,ISSN: 1751-8806.

[74]Maduranga,M and et al. (2016). " DOMAIN SPECIFIC INFRASTRUCTURE FOR CODE SMELL DETECTION IN LARGE-SCALE SOFTWARE SYSTEMS". Sri Lanka: International Research Symposium on Engineering Advancements.

[75]Wangberg,R. (May, 2010). " A Literature Review on Code Smells and Refactoring". Press: Reprosentralen, University of Oslo.

[76] Coding Dojo Blog. (December, 2015). "10 Clean Code Techniques That Every Coder Should Know". http://www.codingdojo.com/blog/clean-code-techniques/. Viewed 8/4/2017.

[77]Computer Hope. (Jun, 2017). "Softwrae Development Process". https://www.computerhope.com/jargon/s/softdeve.htm.viewed27/6/2017.

[78]Wikipedia. (December, 2010). "Softwrae Development Process". https://en.wikipedia.org/wiki/Software_development_process . viewed 25/6/2017.

[79]Expertiza. (December, 2010). "Identifing code smells". http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_517_Fall_2012/ch1b_1w44_as.viewed2/4/2017.

[80]García,F. (Junuary, 2011). " Refactoring Planning for Design SmellCorrection in Object-Oriented Software ". University of Antwerp: SUPERIOR TECHNICAL SCHOOL OF COMPUTER ENGINEERING COMPUTER DEPARTMENT.

[81]Mirza,O. (2007). " Software Performance Evaluation using UML-Ψ (PSI)". Department of Applied Information Technology.

[82]Dobrzaski,L. (July, 2005). " UML Model Refactoring - Support for Maintenance of Executable UML Models ".School of Engineering: Blekinge Institute of Technology.

[83]Käyttöopas. (2016). "Project Analyzer v10 ". http://www.aivosto.com/project/tutorial.pdf.

Appendix A

Restructuring Process

Restructuring must be done systematically to avoid or reduce the risk of introducing bugs on the working code. Martin Fowler wrote a catalogue of 72 restructurings. This section is an extract of that catalogue.

In this appendix; A list of Restructuring Process will be presented which is considered important to elmiminate smells in this theses. A simple explanation is to be provided along with mechanism.

A.1 Rename Method

This Restructuring is not directly related to the duplication removal, but it is used after an extraction in order to name the newly extracted method. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method.

- **Mechanics:**

6. Find a name for the new method you extract.
7. Check to see whether the method signature is implemented by a super-class or subclass. If it is, find another name.
8. Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
9. Compile
10. Change the body of the old method so that a call to the new created one replaces the extracted code.
11. Compile and test.

A.2 Extract Method

If a code fragment can be grouped together, turn it into a method which its name explains the purpose of the method and replace the fragment with a call to the new method.

- **Mechanics:**

1 Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
2 Copy the extracted code from the source method into the new target method.
3 Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
4 See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
5 Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method asit stands.
6 Pass into the target method as parameters local-scope variables that are read from the extracted code.
7 Compile when you have dealt with all the locally-scoped variables.
8 Replace the extracted code in the source method with a call to the target method.
9 Compile and test.

A.3 Parameterize Method

Several methods do similar things, but with different values contained in the method body. We can create one method that uses a parameter for the different values.

- **Mechanics:**

1 Create a parameterized method that can be substituted for each repetitive method.
2 Compile.
3 Replace one old method with a call to the new method.
4 Compile and test.
5 Repeat all the methods, testing after each one.

A.4 Pull Up Method

You have methods with duplicated code on sub classes. You can eliminate the duplication by extracting method from both classes and then by putting it into an upper class in hierarchy. Often Pull Up Method comes after other steps. You see two

methods in different classes that can be parameterized in such a way that they end up as essentially the same method. A special case of the need for Pull Up Method occurs when you have a sub-class that overrides a super-class method yet does the same thing. The most awkward element of Pull Up Method is that the body of the methods may refer to features that are on the subclass but not on the super-class. If the feature is a method, you can create an abstract method in the super-class.

- **Mechanics:**

10 Inspect the methods to ensure they are identical.

11 Create a new method in the super-class, copy the body of one of the methods to it, adjust, and compile.

12 Delete one subclass method.

13 Compile and test.

14 Keep deleting subclass methods and testing until only the super-class method remains.

15 Take a look at the callers of this method to see whether you can change a required type to the super-class.

A.5 Push Down Method

Behaviour on a super-class is relevant only for the subclass. Push Down Method is the opposite of Pull Up Method (See Section A.4).

- **Mechanics:**

1. Declare the method in the subclasses and copy the body.
2. Remove method from super-class.
3. Compile and test.

A.6 Form Template Method

There are two methods in subclasses that seem to carry out broadly similar steps in the same sequence, but the steps are not the same. Move the sequence to the super-class and allow polymorphism to play its role in ensuring the different steps do their things differently. This kind of method is called a template method.

- **Mechanics:**

1. Decompose the methods so that all the extracted methods are either identical or completely different.

2 Use Pull Up Method (see Section A.4) to pull the identical methods into the super-class.
3 For the different methods use Rename Method (see Section A.1) so the signatures for all the methods at each step are the same. This makes the original methods the same in that they all issue the same set of method calls, but the subclasses handle the calls differently.
4 Compile and test after each signature change.
5 Use Pull Up Method on one of the original methods. Define the signatures of the different methods as abstract methods on the super-class.
6 Compile and test.
7 Remove the other methods, compile, and test after each removal.

A.7 Substitute Algorithm

You want to replace an algorithm with one that is clearer. Replace the body of the method with the new algorithm.

- **Mechanics:**

1 Prepare your alternative algorithm. Get it so that it compiles.
2 Run the new algorithm against your tests. If the results are the same, you are finished.
3 If the results are not the same, use the old algorithm for comparison in testing and debugging. Run each test case with old and new algorithms and watch both results. That will help you see which test cases are causing trouble, and how.

A.8 Extract Class

You have one class doing work that should be done by two. Create a new class and move the relevant fields and methods from the old class into the new class.

- **Mechanics:**

1 Decide how to split the responsibilities of the class.
2 Create a new class to express the split-off responsibilities.
   - ✓ If the responsibilities of the old class no longer match its name rename the old class.

3 Make a link from the old to the new class.

- ✓ You may need a two-way link. But don't make the back link until you find you need it.

4 Use Move Field on each field you wish to move.
5 Compile and test after each move.
6 Use Move Method to move methods over from old to new. Start with lower-level methods (called rather than calling) and build to the higher level.
7 Compile and test after each move.
8 Review and reduce the interfaces of each class.
   - ✓ If you did have a two-way link, examine to see whether it can be made one way.

9 Decide whether to expose the new class. If you do expose the class, decide whether to expose it as a reference object or as an immutable value object.

A.9 Extract Super-Class

You have two classes with similar features. Create a super-class and move the common features to the super-class.

- **Mechanics:**

1 Create a blank abstract super-class; make the original classes subclasses of this super-class.
2 One by one, use Pull Up Field (See Section A.15), Pull Up Method (See Section A.4) and Pull Up Constructor Body (See Section A.21) to move common elements to the super-class.
3 Compile and test after each pull.
4 Examine the methods left on the subclasses. See if there are common parts, if there are you can use Extract Method (See Section A.2) followed by Pull Up Method on the common parts. If the overall flow is similar, you may be able to use Form Template Method (See Section A.6).
5 After pulling up all the common elements, check each client of the subclasses. If they use only the common interface you can change the required type to the super-class.

A.10 **Extract Interface**

There is some similarity between Extract Super-class and Extract Interface. Extract Interface can only bring out common interfaces, not common code. Using Extract Interface can lead to smelly duplicate code. You can reduce this problem by using Extract Class to put the behaviour into a component and delegating to it. If there is substantial common behaviour Extract Super-class is simpler, but you do only get to have one super-class.

Interfaces are good to use whenever a class has distinct roles in different situations. Use Extract Interface for each role. Another useful case is that in which you want to describe the outbound interface of a class, that is, the operations the class makes on its server. If you want to allow other kinds of servers in the future, all they need do is implement the interface.

- **Mechanics:**

1. Create an empty interface.
2. Declare the common operations in the interface.
3. Declare the relevant class(es) as implementing the interface.
4. Adjust client type declarations to use the interface.

A.11 **Extract Subclass**

The main trigger for use of Extract Subclass is the realization that a class has a behaviour used for some instances of the class and not for others. Sometimes this is signalled by a type code, in which case you can use Replace Type Code with Subclasses or Replace Type Code with State/Strategy. But you don't have to have a type code to suggest the use for a subclass.

- **Mechanics:**

1 Define a new subclass of the source class.

2 Provide constructors for the new subclass.

   ✓ In the simple cases, copy the arguments of the super-class and call the super-class constructor with super .

   ✓ If you want to hide the use of the subclass from clients, you can use Replace Constructor with Factory Method.

3 Find all calls to constructors of the super-class. If they need the subclass, replace with a call to the new constructor.

- ✓ If the subclass constructor needs different arguments, use Rename Method to change it. If some of the constructor parameters of the super-class are no longer needed, use Rename Method on that too.
- ✓ If the super-class can no longer be directly instantiated, declare it abstract.

4 One by one use Push Down Method and Push Down Field to move features onto the subclass.

- ✓ Unlike Extract Class it usually is easier to work with the methods first and the data last.
- ✓ When a public method is pushed, you may need to redefine a caller's variable or parameter type to call the new method. The compiler will catch these cases.

5 Look for any field that designates information now indicated by the hierarchy (usually a Boolean or type code). Eliminate it by using Self Encapsulate Field and replacing the getter with polymorphic constant methods. All users of this field should be restructured with Replace Conditional with Polymorphism.

- ✓ For any methods outside the class that use an accessor, consider using Move Method to move the method into this class; then use Replace Conditional with Polymorphism.

6 Compile and test after each push down.

A.12 Replace Subclass with Field

You have subclasses that vary only in methods that return constant data. Change the methods to super-class fields and eliminate the subclasses.

- **Mechanics:**

1 Use Replace Constructor with Factory Method (See Section A.17) on the subclasses.

2 If any code refers to the subclasses, replace the reference with one to the super-class.

3 Declare final fields for each constant method on the super-class.

4 Declare a protected super-class constructor to initialize the field.

5 Add or modify subclass constructors to call the new super-class constructor.

6 Compile and test.

7 Implement each constant method in the super-class to return the field and remove the method from the subclass.
8 Compile and test after each removal.
9 When all the subclass methods have been removed, use Inline Method (See SectionA.14) to inline the constructor into the factory method of the super-class.
10 Compile and test.
11 Remove the subclass.
12 Compile and test.
13 Repeat inlining the constructor and eliminating each subclass until they are all gone.

A.13 Decompose Conditional

You have a complicated conditional (if-then-else) statement. Extract methods from the condition, then part, and else parts.

- **Mechanics:**

1. Extract the condition into its own method.
2. Extract the then part and the else part into their own methods.

If I find a nested conditional, I usually first look to see whether I should use *Replace Nested Conditional with Guard Clauses.* If that does not make sense, I decompose each of the conditionals.

A.14 Inline Method

A method's body is just as clear as its name. Put the method's body into the body of its callers and remove the method.

- **Mechanics:**

1 Check that the method is not polymorphic.
2 Find all calls to the method.
3 Replace each call with the method body.
4 Compile and test.
5 Remove the method definition.

A.15 **Pull Up Field**

Two subclasses have the same field. Move the field to the super-class.

- **Mechanics:**

1 Inspect all uses of the candidate fields to ensure they are used in the same way.

2 If the fields do not have the same name, rename the fields so that they have the name you want to use for the super-class field.

3 Compile and test.

4 Create a new field in the super-class.

5 Delete the subclass fields.

6 Compile and test.

7 Consider using Self Encapsulate Field (See Section A.16) on the field.

A.16 **Self Encapsulate Field**

You are accessing a field directly, but the coupling to the field is becoming awkward. Create getting and setting methods for the field and use only those to access the field.

- **Mechanics:**

1 Create a getting and setting method for the field.

2 Find all references to the field and replace them with a getting or setting method.

3 Make the field private.

4 Double check that you have caught all references.

5 Compile and test.

A.17 **Replace Constructor with Factory Method**

When more than one object is created, a simple construction is needed. Replace the constructor with a factory method.

- **Mechanics:**

1 Create a factory method. Make its body a call to the current constructor.

2 Replace all calls to the constructor with calls to the factory method.

3 Compile and test after each replacement.

4 Declare the constructor private.

5 Compile.

A.18 Pull Up Constructor Body

You have constructors on subclasses with mostly identical bodies. Create a super-class constructor; call this from the subclass methods.

- **Mechanics:**

1 Define a super-class constructor.

2 Move the common code at the beginning from the subclass to the super-class constructor.

3 Call the super-class constructor as the first step in the subclass constructor.

4 Compile and test.

A.19 Replace Nested Conditional with Guard Clauses

A method has conditional behaviour that does not make clear the normal path of execution. Use guard clauses for all the special cases.

- **Mechanics:**

1. For each check put in the guard clause.
   - ✓ The guard clause either returns, or throws an exception.
2. Compile and test after each check is replaced with a guard clause.
   - ✓ If all guard clauses yield the same result, use Consolidate Conditional Expressions.

A.20 Collapse Hierarchy

Restructuring the hierarchy often involves pushing methods and fields up and down the hierarchy. After you have done this , you find you have a subclass that is not adding any value, so you need to merge the classes together.

- **Mechanics:**

1 Choose which class is going to be removed: the super-class or the subclasses.

2 Use Pull Up Method (See Section A.4) or Push Down Method (See Section A.5) to move all the behaviour of the removed class to the class with which it is being merged.

3 Compile and test with each move.

4 Adjust references to the class that will be removed to use the merged class. This will affect variable declarations, parameter types and constructors.

5 Remove the empty class.

6 Compile and test.

A.21 **Pull Up Constructor Body**

You have constructors on subclasses with mostly identical bodies. Create a superclass constructor; call this from the subclass methods.

- **Mechanics:**

1. Define a superclass constructor.
2. Move the common code at the beginning from the subclass to the superclass constructor.
3. Call the superclass constructor as the first step in the subclass constructor.
4. Compile and test.

نـهج لتحسين نظم البرمجيات

–بأستحذام الهندسة العكسية و مفهوم إعادة الهيكلة لتحسين جودة شفرة البرنامج

قدمت من قبل

حمزة علي عبدالرحمن القذافي

تحت إشراف

د.توفيق الطويل

الخلاصة

تعتبر رائحة الشفرة أحد العناصر التي تعقّد بشكل كبير قابلية صيانة و تطوير أنظمة البرمجة ، كما يعتبر أحد هياكل البرمجة التي تشير إلي انتهاك مبادئ التصميم الأساسية، مما يؤثر سلبًا على جودة التصميم. في هذا البحث، تم تقديم نهج بسيط يعتمد على تعريفات الهندسة العكسية و مفاهيم إعادة الهيكلة لتحليل و الكشف و إزالة الروائح من شفرة المصدر للانظمة، من خلال تحديد مجموعة من الحالات لهذه الرائحة. يتم تعريف الحالة بالاعتماد علي العلاقة بين الطبقات التي تحتوي على الروائح.

يعتبر النهج المقترح نهجا هجينا، حيث يشتمل علي نوعين من الأساليب أو التقنيات (أسلوب قائم علي الرسم البياني و أسلوب القائم على النص). يستخدم النهج الرسوم البيانية لوصف جميع الحالات المتوقعة لكل رائحه لتسهيل عملية اكتشاف الروائح، بالاضافة لانه يستخدم النصوص لوصف إعادة الهيكلة الفعالة للتخلص من كل حالة. تم تحليل نظام شركة العامة للمطاحن و هذا النظام مطور باستخدام VB.Net 2010 و بعد إضافة التحسينات المناسبة علي النظام باستخدام نهجنا، تم أجراء التحقق الكمي من الصحه النظام قبل وبعد التحسين.

تظهر النتيجة الفعلية للتجربة في دراسة الحالة و هي أن استخدام الحالة يوفر معلومات مفيدة تساعد في اكتشاف رائحة الشفرة، كما يعتبر استخدام الهندسة العكسية فعالا لأنه يدعم الكشف عن الروائح بطريقة بصرية. وأخيرًا ، فإن تعريفات النصية لأعادة الهيكلة المستخدمة تجعل من السهل إزالة الروائح بشكل فعال. بالإضافة إلى أن أستخدام (النقطة 2.3 صفحة 57) ، يجعل القضاء على هذه الروائح يتم بطريقة منظمة، لأنه يصف كل تفاصيل الرائحة و كيفية القضاء عليها. بالإضافة إلى أن التحقق الكمي من الحالة الدراسية قدم نتائج جيدة و التي مكنت الباحث من الاجابة علي كل أسئلة هذا البحث. ولذلك ، فقد حقق هذا البحث أهدافه متمثلة في نهج يساعد المطورين على تحسين شفرة المصدر للأنظمة من خلال التخلص من ثلاثة انواع من روائح الشفرة (الشفرة المتكررة ، الدلالة الطويلة ، الطبقة الكبيرة) وذلك بالاعتماد على مفهوم الهندسة العكسية و مفهوم إعادة هيكلة.

نـهج لتحسين نظم البرمجيات

-بأستحذام الهندسة العكسية و مفهوم إعادة الهيكلة لتحسين جودة شفرة البرنامج

قدمت من قبل:

حمزة علي عبدالرحمن القذافي

تحت إشراف:

د.توفيق الطويل

قدمت هذه الرسالة استكمالا لمتطلبات الحصول على درجة الماجستير في علوم الحاسوب.

جامعة بنغازي

كلية تقنية المعلومات

أبريل 2018