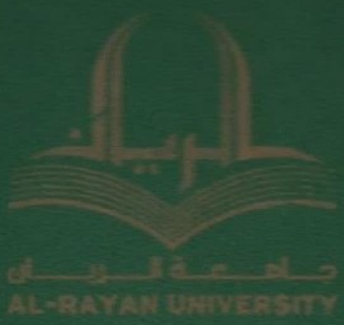
Republic of Yemen
Ministry of Higher Education
& Scientific Research
Al-Rayan University
Faculty of Higher Studies



OPTIMIZING LARGE CLASS SMELL BY APPLYING CLASS NORMALIZATION RULES

Thesis Submitted to Al-Rayan University to Fulfillment of the Requirements for the Degree of Master in Information Technology

By

Marwan Ahmed Saeed Lardhi

Supervised by

Dr. Saeed Mohammed Baneamoon

1441 / 2020

Republic of Yemen

Ministry of Higher Education
& Scientific Research

Al-Rayan University

Faculty of Higher Studies

OPTIMIZING LARGE CLASS SMELL BY APPLYING CLASS NORMALIZATION RULES

Thesis Submitted to Al-Rayan University to Fulfillment of the Requirements for the Degree of Master in Information Technology

By

Marwan Ahmed Saeed Lardhi

Supervised by

Dr. Saeed Mohammed Baneamoon

1441 / 2020هـ

Approval of the Proofreader

I certify that the master's dissertation titled,

(**Optimizing Large Class Smell by Applying Class Normalization Rules**)

submitted by the student, **Marwan Ahmed Saeed Lardhi**

has been linguistically reviewed under my supervision and has become in scientific style and clear from linguistic errors and for that I sign.

Proofreader : Abdullah Amer Al-kathiri

Academic Title : Assistant Professor

University : Hadhramout University

Signature : .....

Date : 29 / 6 /2020

Approval of the Scientific Supervisor

I certify that this master's dissertation titled,

(**Optimizing Large Class Smell by Applying Class Normalization Rules**)
submitted by the student, **Marwan Ahmed Saeed Lardhi**

has been completed in all its stages under my supervision and so I nominate it for discussion.

Supervisor : Dr. Saeed Mohammed Baneamoon
Signature : .....
Date : 29 / 6 /2020

The Discussion Committee Decision

Based on the decision of the President of the University No. (4) in the year 2020 regarding the nomination of the committee for discussing the master's thesis entitled (**Optimizing Large Class Smell by Applying Class Normalization Rules**) for the researcher **Marwan Ahmed Saeed Lardhi**. We, the head of the discussion committee and its members, acknowledge that we have seen the aforementioned scientific thesis and we have discussed the student in its contents and what related to it.

Chairman of the Committee

Associate Professor Dr. Khalid Kayid Shafel

Signature: ..... 13.1.2021 .....

Committee member

Associate Professor

Dr. Saeed Mohammed Baneamoon

Signature:.....

Committee member

Assistant Professor

Dr. Mohammed Abdullah Bamatraf

Signature:.....

قال الله تعالى:-

أَمَّنْ هُوَ قَٰنِتٌ ءَانَآءَ ٱلَّيْلِ سَاجِدًا وَقَآئِمًا يَحْذَرُ ٱلْءَاخِرَةَ وَيَرْجُواْ رَحْمَةَ رَبِّهِۦ ۗ قُلْ هَلْ يَسْتَوِى ٱلَّذِينَ يَعْلَمُونَ وَٱلَّذِينَ لَا يَعْلَمُونَ ۗ إِنَّمَا يَتَذَكَّرُ أُوْلُواْ ٱلْأَلْبَٰبِ ﴿٩﴾

[سورة الزمر،٩]

Dedication

I dedicate this work to

My father … My mother

and

My wife … My daughters

who made this accomplishment possible.

Acknowledgement

First, I present all thanks to Allah for all thing and absolute helpness.

Second, I would like to thank my supervisor Dr. Saeed Mohammed Baneamoon who has provided me with support and guidance in this thesis.

And last but not least, I would like to give thanks and the love to my family who has supported me always towards my goals and ambitions.

Abstract

Software needs to be updated and modified after delivery to correct faults and enhance system quality. Classes of system undergoes continues modifications and making that source code complex and difficult to maintain. As a result, the class becomes large and called in this case large class smell. This study proposes an effective method for optimizing extraction large class smell using class normalization rules in order to ease maintenance and improve the quality of software by creating new classes with strongly and similarity attributes and shared behavior. The proposed method introduced a technique to extract a class with many responsibilities and that is chosen by the developer or automatically, where is produced an access-set table of attributes, and then is calculated the Jaccard similarity measure to create a similarity matrix for attributes. After that is designed the structural similarity matrix of each extracted class, to calculate the cohesion of each class. Experimental results show that applying the proposed method for dividing the large class into many cohesive classes provide better performance in software development compared with existing methods.

الملخص

تحتاج البرمجيات إلى التحديث والتعديل بعد التسليم لتصحيح الأخطاء وتحسين جودة النظام. حيث تخضع فئات النظام للتعديلات المستمرة وتجعل الشفرة المصدرية معقدًة ويصعب صيانتها نتيجة لذلك ، تصبح الفئة كبيرة ويطلق عليها في هذه الحالة فئة كبيرة. تقترح هذه الدراسة طريقة فعالة لتحسين استخراج الفئة الكبيرة باستخدام قواعد تطبيع الصف من أجل تسهيل الصيانة وتحسين جودة البرنامج من خلال إنشاء فئات جديدة ذات سمات تشابه قوية وسلوك مشترك. قدمت الطريقة المقترحة تقنية لاستخراج فئة لها العديد من المسؤوليات التي يتم اختيارها من قبل المطور أو تلقائيًا ، حيث يتم إنتاج جدول سمات الوصول ، ومن ثم يتم حساب مقياس تشابه الجاكارد لإنشاء مصفوفة تشابه للسمات (المتغيرات). بعد ذلك يتم تصميم مصفوفة التشابه الهيكلي لكل فئة مستخرجة لحساب تماسك كل فئة. تظهر النتائج التجريبية أن تطبيق الطريقة المقترحة لتقسيم الفئة الكبيرة إلى العديد من الفئات المتماسكة يوفر أداء أفضل في تطور البرمجيات مقارنة بالطرق الحالية.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Symbols	Nomenclatures
1ONF	First Object Normal Form
2ONF	Second Object Normal Form
3ONF	Third Object Normal Form
SDLC	Software Development Life Cycle
IEEE	Institute of Electrical and Electronics Engineers
OO model	Object Oriented model
C3	Conceptual Cohesion of Classes
ESR	Extract Subclass Refactoring
FRC	Functional over-Related Classes
UML	Unified Model Language
RePOR	Refactoring approach based on Partial Order Reduction
GA	Genetic Algorithm
ACO	Ant Colony Optimization
SSM	Structural Similarity between Methods
ClassCoh	Class Cohesion

CHAPTER 1: INTRODUCTION

1.1 Overview

Maintenance of software is a component of the life cycle of software development that the primary aim is to modify and update software application after delivery to correct faults and enhance system efficiency where it could be easy to modifications to correct coding mistakes, more comprehensive modifications to correct design mistakes, or to accommodate new requirements (Sommerville, 2016). There are things that impair software quality and make them hard to maintain and evolve like code smells. Code smell is signs inside the code that indicate that there is a design flaw and is not in a software error, where it may find codes full of these smells but they work just fine without any problems.

To improve software maintainability, there are several refactoring techniques that may apply to source code. Refactoring is a change made to the software's inner structure to make it simpler to comprehend and cheaper to change without altering its behavior such as Move Field, Move Method, Extract Method, Pull Up Field and Extract Class. First, refactoring improves software design where changes to realize short-term goals or changes made without a full understanding of the code's design the code loses its structure, making it more difficult to see the design by reading the code and the poorly designed code which usually requires more code to do the same things. Second, it makes software easier to understand, programming is a write code conversation with a computer that informs the computer what to do, and it reacts by doing precisely what you say and programming in this mode is all about stating precisely what you want, but somebody will attempt to read this code. But there's another user of this source code which in a few months' time someone will attempt to read code to create some changes, which means that additional code user can readily be forgotten. Third, refactoring helps to find bugs, since understanding the code can help identify bugs that some can read a bunch of code and see bugs. Lastly, it helps with programming quicker where the whole point of getting a good design is to enable fast development and without a good design can the progress rapidly for a while, but soon the bad design starts slowing down the developer and thus spend time finding and fixing bugs instead of adding a new feature where modifications take longer as an attempt to comprehend the system and discover the duplicate code (Fowler & Beck, 1999).

Classes usually start small, but over time they become larger as the software expands. As is the case for long methods, programmers usually find it less exhausting mentally to put a new feature in an existing class than to create a new class for the feature and important to improve any program's structure, maintenance and improve performance where refactoring is the key to improve both the quality of the code (Fowler & Beck, 1999). The extract class refactoring method will help maintain adherence to the single responsibility principle and classes are more reliable and tolerant of changes.

Class normalization techniques are not yet as popular as refactoring or pattern application. Class normalization is a process through which object schema structure is reorganized in such a way that class cohesion is increased with coupling is minimized between classes. The Repeating data structures are refactored into their own class to place a class in the first object normal form (1ONF). When encapsulating the shared behavior required by multiple entities within its own class, a class is in the second object normal form (2ONF). A class is in the third object normal form (3ONF) when implementing a single, cohesive set of behaviors (3ONF) (Ambler, 2003). Hence, this study will focus on the challenge of using the class normalization rules to refactoring large class and extracting into classes with a high coherence and a specific behavior.

1.2 Motivation

Programmers may encounter software that were difficult to maintain and develop. Often the original developer of this kind of software had left the organization or continuous modifications making the source code more complex and drifting away from its original design and becomes complex and difficult maintenance, leaving other developers to deal with the situation that detecting badly structured code. Also, one of the key messages of software evolution is that software products need to be continuously modified to maintain the competitive edge and main point is that continuous modification makes software structure more complex, unless effort is made to reduce this complexity. One way to reduce this complexity is refactoring, which aims to improve the software structure without changing the software behavior and preventing these problems and benefit the working environment of software developers, and help organizations by making their software development more productive.

1.3 Problem Statement

Classes of system undergo continuous modifications making the source code more complex and drifting away from its original design so the class becomes very complex, multi-responsibility, difficult to maintain and its quality deteriorates. This problem is known a large class that is one of the code smells where a class with too much code is prime breeding ground for duplicated code and chaos.

1.4 Objectives

This research aims to improve extract large class smell by applying class normalization rules, where the large class is one of the code smells that increase difficult of software maintenance, as the system needs to be constantly modified and developed after testing or delivery, either by fixing errors or adding new features. Therefore, the presence of a class or classes full of variables and methods with different behavior makes the system environment chaotic and disorganized.

However, this research addresses some issues in software development. The first issue is the degraded and unarranged system structure. The second issue is costs and effort of Maintenance . In more detail, the objectives are:

- To decrease the large class by separating it into classes with strongly related and more similarity attributes and behaviors in order to increase the focus of class because the more that code in a class supports a central purpose, the more easily knowing everything the code does.
- To evaluate the conformity of the extracted classes with the class normalization rules in order to determine these all classes or one are identical or need to enhance and refactoring.

1.5 Scope and Limitations

The scope of study in software engineering and development which was built on Microsoft .NET technologies, specially C# .Net language, that based on refactoring large class smell to smaller classes according to class normalization rules. The large class would be extracted according to similarity and cohesion of variables and methods. Although refactoring could be used with any development and applied on other code smells which identified M. Fowler in his book (Fowler & Beck, 1999), but this study is limited on the large class smell and how treated with it.

1.6 Research Approach

In order to perform the objectives of this research, the steps involved in this research are as shown in Figure (1.1).

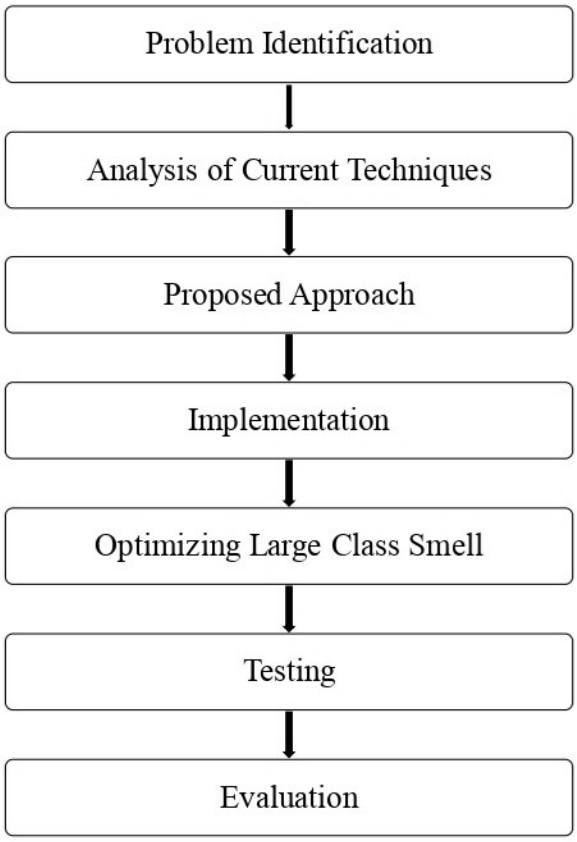


Figure (1.1) Diagram of Research Approach

1.6.1 Problem Identification

This problem is known as a large class smell which is one of the smells of code where system classes will undergo continuous modifications which make the source code more complex and ground for chaos and drift away from the original design.

1.6.2 Analysis of Current Techniques

A number of studies will be analyzed to know the current techniques used in the code refactoring and improving the smell of the large class such as, the design pattern, code reorganization. The result will be shown how these techniques work and what distinguishes each one from the other and the strengths and weaknesses of each, and the most important limitations that this study try to solve.

1.6.3 Proposed Approach

The proposed method for extracting large class by simulating the three rules for normalizing classes (1ONF, 2ONF and 3ONF). The method boils down to take a class with many responsibilities , after that produce an access-set table of attributes, and then calculating the Jaccard similarity index to create a similarity matrix for attributes. The structural similarity matrix will be created to compute the cohesion of each class, thus achieving the third rule for normalization, so the class is behaviorally coherent.

1.6.4 Implementation

The study will be applied to the C#.NET language of Microsoft. NET technologies. Study is based on extraction of large class odor to smaller classes according to the rules of class normalization. The large class will be extracted according to the similarity of variables and methods and their cohesion. The method boils down to taking a class with many responsibilities nominating for extraction, where the parser produces an access-set table of attributes, then the similarity matrix of the attributes will be calculated by calculating the similarity ratio of Jaccard which measures the similarity between two sample sets. Thus, the matrix for similarity has values in [0 , 1]; where the value of 1 for a number of attributes indicates that it is in the same class with the methods to which it relates. As a result of the original class extraction, a number of classes proposed are consisting. The structural similarity matrix will be created by summing the similarities of all method pairs and dividing by the total number of such pairs to compute the cohesion of each class. And according to results of calculating this metric, extracted classes will be evaluated as coherent or incoherent and requires re-extraction.

1.6.5 Optimizing Large Class Smell

A method to optimize extraction of large class smell using class normalization rules to ease maintenance and improve software quality by creating new classes with strong and similarity attributes and shared behavior.

1.6.6 Testing

In this stage, will be tested the proposed method on the C# class which has a set of operations that manipulate the user entity in the database such add a user, editing user, etc. However, this class contained two other responsibilities, i.e., the Teaching

Entity management and the Role Entity management. The required is to separate this class so that each entity becomes in a separate class and with a specific responsibility of defining single responsibility methods in the class. The results will be shown the ability of the proposed method to extract the class, where the class was separated into three classes, each containing the code for its tasks.

1.6.7 Evaluation

The method will show the importance of refactoring to enhance class quality and simple the maintenance, where it improves class structure, makes more organization and provides better performance in software evolution compared with existing methods, and increases the size of software to increase the number of classes from its disadvantages.

1.7 Research Contributions

The extract class refactoring technique introduced in this study evaluates the cohesiveness of a given class and suggests re-factoring the class by extracting new classes which coherence and shared behavior of these classes will be shown. Also, the importance of the research that set class normalization rules as criterion of extracting large classes for specifying its cohesion, behaviors and responsibility.

1.8 Structures of Thesis

This thesis is organized into five chapters. Chapter 2 provides necessary terminology, concepts and background regarding the subject and to understand the rest of thesis. Chapter 3 shows the most important literature and related work illustrating the importance of refactoring and the proposed approaches of extracting the large class, showing its mechanism, strengths and limitations. Chapter 4 details of the design and implementation of the proposed approach to extract the large class, experimenting with it, and showing and discussing the results of the experiment. Finally, Chapter 5 presents a conclusion of the study and proposes what should be done as future work.

CHAPTER 2: BACKGROUND

2.1 Overview

Software engineering is an engineering branch associated with development of software product using scientific principles, methods and procedures. This chapter gives an introduction about software engineering and its importance for building software. And reviews the maintenance phase, which is one of the stages of SDLC, explaining its importance, types and cost impact in the SDLC, clarifying the difference between it and software reengineering. Refactoring is a change makes to the software's inner structure to make it simpler to comprehend without altering its behavior. This chapter introduces the refactoring, how it to affect on the software system and lack of influence on the behavior of the system. Then, presents the code smells, and how to affect the system code and classes, explaining briefly their types and what possible solutions for each. Also, it gives an extensive explanation of the large class that is the subject of this research, clarifying what it, how arises and the best solution to extract and improve it. In end explains, class normalization technique that is a process through which object schema structure is reorganized in such a way that class cohesion is increased with coupling is minimized between classes.

2.2 Software Engineering

Software engineering is an engineering discipline that covers all aspects of software development from initial design through to operation and maintenance, where software is not just an app or programs, furthermore, it also includes all the digital documentation required by system users, quality assurance personnel and developers to ensure that security, reliability and safety, efficiency and acceptability are key technology features and are relevant and important for two reasons. The first of them, individuals and society depend on advanced software systems, so reliable and trustworthy systems need to be developed economically and rapidly. The second is that it is usually cheaper to use software engineering methods and techniques for professional software systems in the long run rather than just writing programs as a personal programming project (Sommerville, 2016).

2.3 Software Maintenance

The IEEE defines software maintenance in their *IEEE standard for Software Maintenance (IEEE 1998)* as:

''*Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*''

Also, software maintenance stands for all the modifications and updates done after the delivery of software product. There are a number of reasons, why modifications are required, some of them are as follows (*Software Engineering Tutorial*, 2014):

- Market conditions where policies, which changes over the time, such as taxation and newly introduced constraints may trigger need for modification.
- Client Requirements who may ask for new features or functions in the software.
- Host modifications where if any of the hardware and/or platform (such as operating system)of the target host changes, software changes are needed to keep adaptability, software changes are needed to keep adaptability.
- Organization changes that if there is any business level change at client end, such as reduction of organization strength, acquiring another company, need to modify in the original software may arise.

A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.

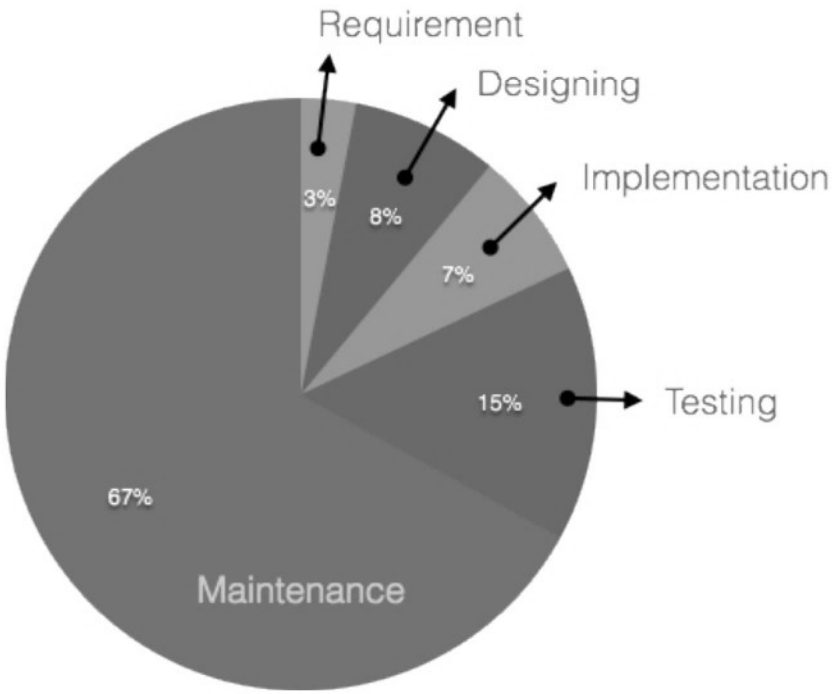


Figure (2.1) Software Life Cycle Cost

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are three types of software maintenance (Sommerville, 2016):

1. Fault repairs: Coding errors are usually relatively cheap to correct, while design errors are more expensive because they may involve rewriting several program components.
2. Environmental adaptation to adapt the software to new platforms and environments where this type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software.
3. Functionality addition to add new features and to support new requirements where this type of maintenance is necessary when system requirements change in response to organizational or business change.

2.3.1 Software Reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, to make legacy software systems easier to maintain, we can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and has two important advantages, one of them is reduced risk and another, reduced cost (Sommerville, 2016).

The nexus between business reengineering and software engineering lies in a "system view." As managers work to modify business rules to achieve greater effectiveness and competitiveness, software must keep pace. In some cases, this means the creation of major new computer-based systems. But in many others, it means the modification or rebuilding of existing applications (Pressman, 2010).

2.4 Refactoring

The Cardinal law of software evolution is that evolution will boost a program's internal efficiency (McConnell, 2004), and one of the primary techniques is to refactor. The primary objective of refactoring is to improve current code design. Without refactoring, as a result of successive modifications and extensions, the program's design will automatically decay.

Refactoring is process of changing a software system in such a way that is not done alter the external behavior of the code yet improves its internal structure (Fowler & Beck, 1999).

Code refactoring is a disciplined technique to restructure an existing code object, modifying its internal structure without altering its external actions, performed to enhance some of the software's non-functional attributes. This is typically done by applying a series of "Refactoring," each of which is (usually) a tiny change in the source code of a computer program that does not change its compliance with functional requirements. Advantages include increased readability of code and decreased complexity to enhance source code maintenance, as well as a more flexible internal architecture or object model to enhance extensibility (Kaur & Kaur, 2016).

The importance of refactoring lies in several reasons, as clarified by Fowler (Fowler & Beck, 1999) as follows:

- *Making software easier to understand*. Programming is a write code conversation with a machine that tells the computer what to do, and it responds with what you're doing, because one of the aims of refactoring is that software would be easy to understand and source code would be self-documenting, making it easier for another user to understand and alter this source code.
- *Helping to find bugs*. Understanding the code will help identify bugs that some people can read a lot of the code and see bugs, and refactoring will allow programmers to write stable code much more effectively.
- *Improving the design of software*. This argument goes hand in hand with the first argument, since good design is easy to understand where a change made without a complete understanding of the design of the code loses its structure and requires more code to do the same things.

- *Helping program faster*. This argument is supported by the software evaluation laws according to which raising the complexity of the software system will impede the development of software where the whole point of getting a good design is to enable fast development and without a good design can advance quickly for a while, but soon the bad design begins to slow down developers.

2.5 Code Smells

A code smell is a surface indication that typically corresponds to a system's deeper problem. There are a couple of points in this fast description. First, a smell is something that is easy to detect-or sniffable by definition. A long method is a good example-just look at the code. The second is that there is not always a problem with smells. Most long strategies are all right. You have to look deeper to see if there is an underlying problem there-odors on their own are not inherently bad-they are often a symptom of a problem rather than the problem itself (Fowler & Beck, 1999).

Code smells are implementation structures that adversely affect system lifecycle properties such as comprehensibility, testability, extensibility and reusability; that means, code smells ultimately leads to maintenance problems. So, smells in software systems can affect the quality of software and make it difficult to maintain and develop. There are several types of code smells that can be classified into units, each of which contains a group of types of smells and the following table describes and refactors methods of them (Singh & Kaur, 2017):

Table (2.1) Code smells Types

No	Unit	Smell	Description	Solution
1	Bloaters	Long Method	A method has many lines of code. Generally speaking, any method longer than ten lines should cause you to begin asking questions.	Extract Method, Replace Temp With Query, Introduce Parameter Object and Preserve Whole Object. Replace Method
		Large Class	A class contains a lot of code fields, methods, or lines.	Extract Class, Extract Sub Class
		Data Clump	Often different parts of the code contain similar groups of variables (such as parameters for connecting to a database).	Extract Class, Introduce Parameter Object and Preserve Whole Object
		Primitive Obsession	Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)	Replace Data Value With Object, Replace Type Code With Class, Replace Type Code With Subclass., Replace Type Code With State Strategy, Extract Class, Introduce Parameter Object And Replace Array With Object.
		Long Parameter List	For a method, there are more than three or four parameters.	Replace Parameter with Method
2	Object Orientation Abusers	Switch Statements	There's a complex turn operator or if statement.	Extract Method, Move Method, Replace Type Code with Subclass or Replace Type Code with State
		Temporary Field	Temporary fields only get their values under certain conditions (and are therefore needed by objects). They are empty outside these conditions.	Extract Class, Introduce Null Objects
		Refused Bequest	When a subclass only uses some of its parents ' inherited methods and resources, the hierarchy is off-kilter. Unneeded methods may either go unused or be redefined to make exceptions available.	Push Down Method Push Down Field, Replace Inheritance with Delegation
		Alternative Classes with Different Interfaces	Two classes do the same functions, but they have different names of methods.	Rename Method, Move Method, Extract Superclass
3	Change Preventers	Divergent Change	When making changes to a class, often unrelated methods need to be modified.	Extract Class
		Shotgun Surgery	This smell occurs when there is one type of change that results in many changes to several different classes.	Move Class or Move Method
		Parallel Inheritance Hierarchies	The need for a subclass to be generated for another class when constructing a subclass.	Move Method or Move Field

Table (2.1) Code smells Types (Cont'd)

No	Unit	Smell	Description	Solution
4	Dispensables	Lazy Class	It happens when a class was intended to be fully functional but after some of the refactoring it has become ridiculously small or it was designed to support future development work that never got done.	Collapse Hierarchy or Inline Class
		Data Class	This smell happens when a class includes only fields and likely getters/setters without any actions (methods).	Encapsulate Field or Encapsulate Collection, Remove Setting Method, Move Method or Extract Method, Hide Method
		Duplicate Code	Duplication usually occurs as several programmers operate on different parts of the same software at the same time. Since they are operating on different tasks, they may be unaware that their friend has already developed a similar code that could be reworked for their own purposes.	Extract Method, Pull Up Method, Substitute Algorithm.
		Dead Code	A variable, parameter, field, method or class is no longer used when program specifications have modified or changes have been produced and no time has elapsed to clean the old code and is also contained in complicated conditionals where one of branch is unreachable.	Collapse Hierarchy, Inline Class, Remove Parameters
		Speculative Generality	There is an unused class, method, field or parameter where code is sometimes created to support the expected future features that are never implemented. As a result, the code is difficult to understand and support.	Collapse Hierarchy, Inline Class, Remove Parameters Methods, Remove Methods
5	Encapsulators	Message Chains	A message chain occurs when a client requests another entity; the object requests another one, and so on. Such chains mean that the client relies on moving along the class structure.	Hide DELEGATE, Extract Method and Move Method.
		Middle Man	This smell can be the product of the overzealous removal of Message Chains, where a class executes only one operation, which delegating function to another class.	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
6	Couplers	Feature Envy	A method has more access to the data of another object than its own data. Where it may occur after the fields have been moved to the data class. In this case, there is a need to move data operations to this class as well.	Move Method and Extract Method
		Inappropriate Intimacy	This smell happens when a one class uses the inner fields and methods of another class.	Move Method or Move Field, Change Bidirectional Association to Unidirectional, Extract Class, Hide Class
7	Others	Incomplete Library Class	When libraries stop meeting the needs of the user. The only solution to the problem is to change the library.	Move Method, Introduce Foreign Method, Introduce Local Extension
		Comments	When the author discovers that his or her code is not intuitive or apparent, comments are usually made. Comments in such situations are like a deodorant that hides the odor of fishy code that can be changed.	Extract Method or Rename Method, Introduce Assertion.

2.6 Large Class

The large class is one of code smells that contains many fields, methods or lines of code. As with a class with too many instance variables, a class with too many codes is prime breeding ground for duplicated code, chaos, and death. The simplest solution is to eliminate redundancy in the class itself. If you have five hundred-line methods with lots of codes in common, you may be able to turn them into five ten-line methods with another ten two-line methods extracted from the original (Fowler & Beck, 1999).
Traditional way of measuring class size has been to measure the number of attributes and methods, but the best measure of class size is class cohesion. Class cohesion metrics such as Lack of Cohesion Methods can sometimes be difficult to calculate. So in such cases, the number of methods and attributes should be used as a measure of class size.

Refactoring of these classes that splitting large classes into parts avoids duplication of code and functionality and the following methods using to split:

- *Extract Class*. This helps if part of the behavior of the large class can be spun off into a separate component, e.g. having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle as shown in Figure (2.2).

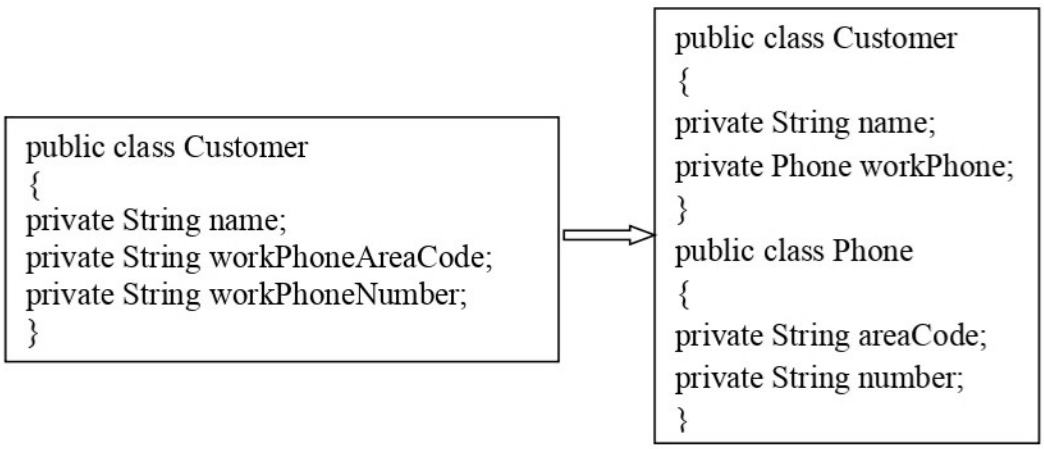


Figure (2.2) Extract class

- *Extract Subclass*. The extract helps if part of the behavior of the large class can be implemented in different ways or is used in rare cases. When a class has features (attributes and methods) that would only be useful in specialized

instances, can create a specialization of that class and give it those features as shown in Figure (2.3).

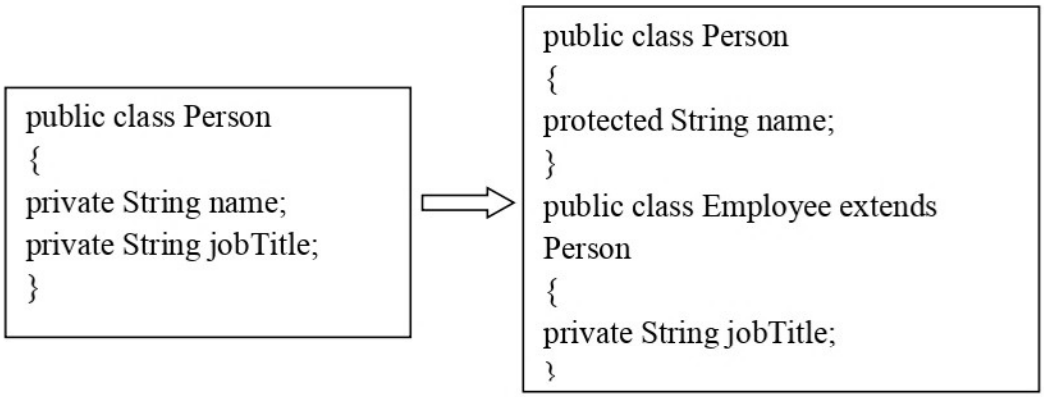


Figure (2.3) Extract subclass

- *Extract Interface.* This helps if it is necessary to have a list of the operations and behaviors that the client can use as shown in Figure (2.4).

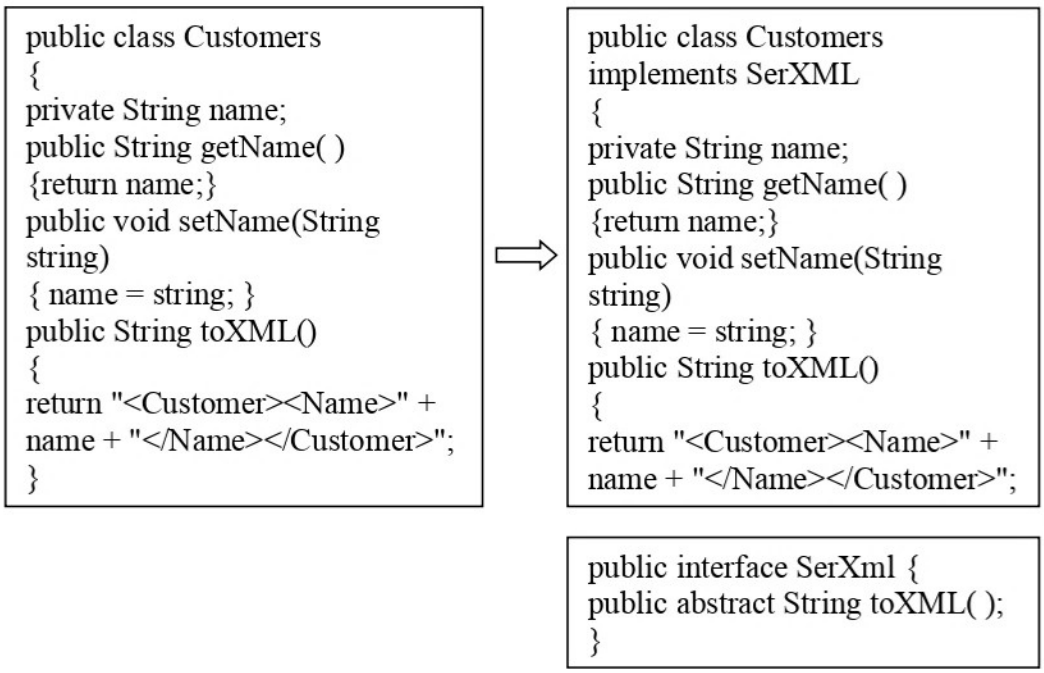


Figure (2.4) Extract interface

2.7 Class Normalization

Class normalization is a technique or process by which we reorganize the structure of object schema in such a way as to increase the cohesion of classes while minimizing the coupling between them for improving the quality of object schemas. Cohesion is the degree to which the aspects of an encapsulated unit (such as a component or a class) are

related to one another. While coupling is the degree of dependence between two items (Ambler, 2003).

2.7.1 Class Normalization Rules

The rules of class normalization are discussed and how these works, further in the following sections.

a. First object normal form (1ONF)

Suppose the class in Figure (2.5) that needs to normalize and can say class is in 0ONF. A class is in 1ONF when specific behavior required by an attribute that is actually a collection of similar attributes is encapsulated within its own class as shown in Figure (2.6).

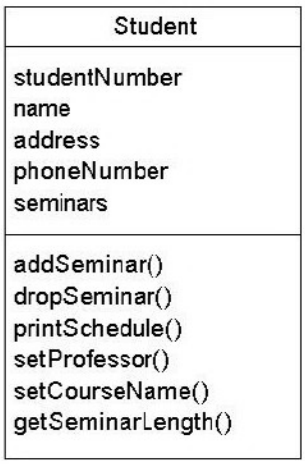


Figure (2.5) 0ONF

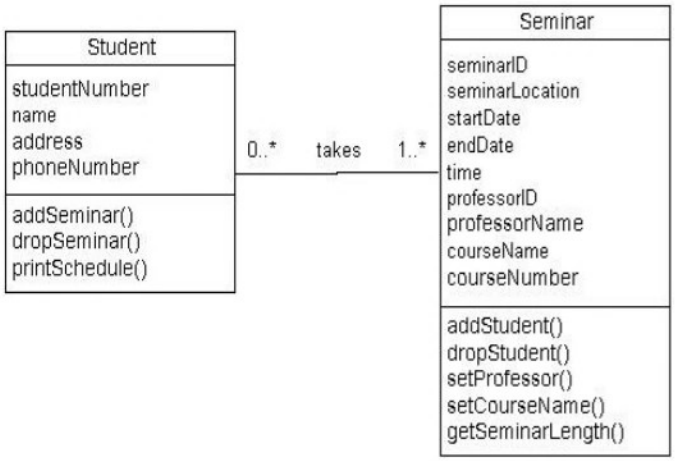


Figure (2.6) 1ONF

b. Second object normal form (2ONF)

A class is in 2ONF when it is in 1ONF and when "shared" behavior required by more than one instance of the class is encapsulated within its own class(es) as shown in Figure (2.7).

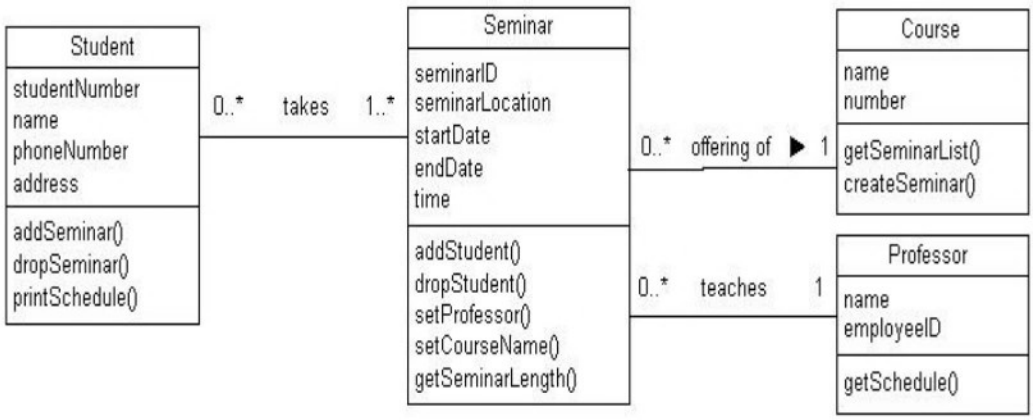


Figure (2.7) 2ONF

c. Third object normal form (3ONF)

A class is in 3ONF when it is in 2ONF and when it encapsulates only one set of cohesive behavior as shown in Figure (2.8).

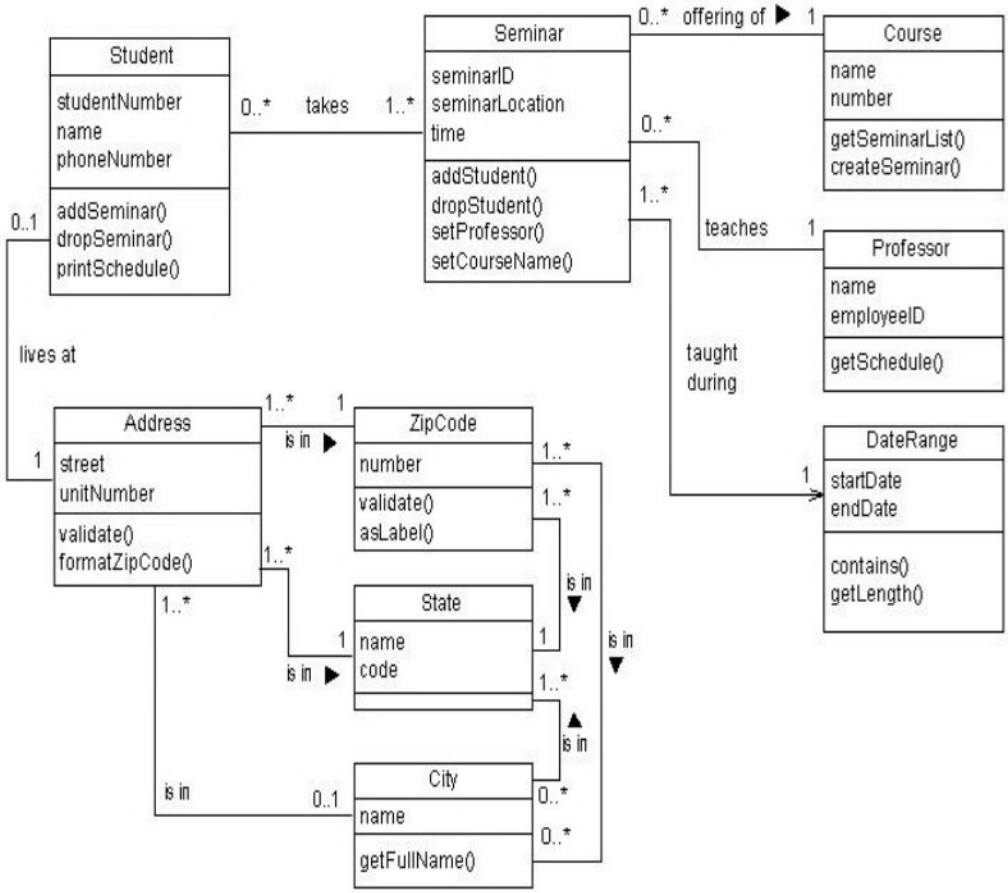


Figure (2.8) 3NOF

2.7.2 Class Normalization and Refactoring

Class normalization and refactoring fit together quite well-as normalizing classes will effectively be applying many known refactorings to object. Whereas a fundamental difference between class normalization and refactoring is that class normalization is performed to class models and refactoring are applied to source code. Although the techniques of class normalization aren't yet as popular as refactoring or the application of design patterns, that they are important because they provide a very good bridge between the object and data paradigms.

2.8 Remarks

This study focuses to extract the large class smell by applying the rules of normalization of classes (1ONF, 2ONF and 3ONF). Class normalization is a process by which you reorganize the structure of object in such a way as to increase the cohesion of

classes where extract class by applying these rules to split into new class(s). Which will contribute to organize the structure of the code and, thus facilitate the maintenance phase in the program life cycle.

2.9 Summary

This chapter provided an overview into the software engineering and maintenance that clarifies what maintenance represents from the cost of the software life cycle. And code smells have an impact on the program's maintenance process, although they have no clear effect on the program's work, but affecting in the program's structure. Reviewed the solutions which contribute to solve the code smells where explained the refactoring and importance and types, and the most important solutions provide for each type of smells. Presented details for the large class smell, which is the subject of our study and how can solve with several methods. Also, gave a definition of the rules of normalization of classes, importance and relationship to refactoring.

CHAPTER 3: LITERATURE REVIEW

3.1 Overview

A number of studies were conducted for bad smells of programming codes and various authors have studied the impact of refactoring of the source code. This chapter, shows studies and researches that all agreed on the effect of the code smells on the quality and maintenance of the software, some of them presented methods to detect and others focused on how to apply methods for refactoring of code.

3.2 Related Works

Marcus et al. (2008) proposed a new Object-Oriented (OO) software class cohesion measure based on the analysis of unstructured information embedded in the source code, which called the Class Conceptual Cohesion (C3) such as comments and identifiers, where structural and conceptual metrics are combined to provide better models for classes faults prediction than combinations of structural metrics alone. In this approach it was noted that the approach does not take into account polymorphism and inheritance, and its reliance on the existence of naming's conventions for the relevant identifiers and comments, and when these are missing, the effect on measuring the coherence of the class appears.

Bavota et al. (2011) proposed a method based on graph theory that exploits structural and semantic relationships between methods in a class to be refactored that to build two new classes having higher coherent than the original class. Bavota et al. (2014) came back to update of the previous work, where they presented a method chains used to define new classes with a higher coherent than the original class, while preserving the overall coupling between the new classes and the classes that interact with the original classes. This study distinguished its ability to increase the strength of cohesion of classes without a significant increase in coupling, but relying on generalized results from master's sample experience poses a threat.

Chu et al. (2012) presented test case refactoring for extreme programming which leads to faster development. Two patterns from the Gang of Four were selected for the application of refactoring and the validation of study. The approach not an automatic completely, where the refactored application code or test cases cannot be generated automatically, because refactorings are applied to the whole refactoring process frequently.

Al Dallal (2012) proposed a model that was applied to automatically predict the classes that need ESR and present them as suggestions for developers working to

improve the system during the maintenance phase, as the models created using studied quality metrics showed high capabilities to separate the classes in those that were in need and those that did not need to resell housing. This model was slow and time consuming, because does a complete scan of the code and analyzed the relationships between the layers to determine the classes.

Fokaefs et al. (2012) introduced a method accompanied by tool-based for identifying source code chunks which collaborate to provide a particular job and propose extraction as detach methods. The proposed work identified the design defects with the Eclipse plug-in which affected coupling and cohesion. Suggestions could have been better and more complete if the clustering algorithm was combined with other methods, like code duplication detection techniques.

Dexun et al. (2014) suggested that classes that were not functionally related could generate software maintenance problems, hence the detection and refactoring of such classes was necessary. The basic process is to gather the dependence relationships between classes, calculate the invoking rates and compare them with dynamic threshold. But the thresholds in FRC bad smell detection that are preset thresholds decrease the veracity of detection results.

Bavota et al. (2015) presented an experiment aimed at investigating the characteristics of code components increasing their changes of being subject to refactoring operations where was verified whether refactoring activities occur on classes for which indicators might indicate to be needed for refactoring, such as quality metrics or the presence of smells as detected by the tools-suggest. Quality metrics have not demonstrated a clear relationship with refactoring in some cases, where metrics may not be per se indicators of smells.

Kaur & Kaur (2016) used Eclipse tool to refactor the bad smells and make an easy source code to understand. The complexity of the project was then calculated and compared with the initial complexity, and the results were checked. This study confirmed the importance of refactoring that makes a code easier to understand and improve the quality and reduce the maintenance cost.

Zafeiris et al. (2017) proposed a method for automated refactoring to the template method design pattern of certain design flaws related to concrete method overriding, where an overriding method includes in its body an invocation to the overridden method through the super keyword (super-invocation), then applied the Template Method design pattern for the elimination of appropriate Call Super instances from a code base,

which introduced an algorithm for the discovery of refactoring opportunities based on a broad set of preconditions for the refactoring. Consequently, the results of this study cannot be generalized to a project or projects written in another programming language other than java.

Morales et al. (2018) presented a novel approach for automatically scheduling refactoring operations for correcting anti-patterns in software systems where conducted a case study with five open-source software systems and compared the performance of RePOR with the performance of two well-known metheuristics (GA and ACO), one conflicting-aware refactoring approach (LIU), and a recent metaheuristic based on sampling (Sway). Results showed that RePOR can correct more anti-patterns than the techniques in just a fraction of the time, and with less effort. However, was compared with genetic algorithm which, is known has computationally expensive, i.e. time-consuming, so this poses a threat for results of the approach.

Turkistani and Liu (2019) designed a method for dealing with the Large Class problem by classifying the causes of the code smell and applying different design patterns to refactor the code to improve the quality of the software, analyzing the causes of the Large Class code smell and classifying them into corresponding types and proposing a design pattern to address each type to refactor the code. So approach helped to refactor the code to make the maintenance, modification, and reusable easy.

Mooij et al. (2020) presented an exploratory case study that aimed to rejuvenate an industrial embedded software component implementing a nested state machine. Where develop and apply a series of small, automated, case-specific code refactorings that ensure the code uses well-known programming idioms, then perform model-based rejuvenation focusing on the high-level structure of the code. And therefore gives ample opportunity to be validated early in the form of code reviews and testing since each refactoring is carried out directly on the existing code. Moreover, aligning the code with the type of model simplifies the extraction, making the process less error-prone.

3.3 Critical Evaluation on Existing Approaches

The previous literature contained a number of limitations where some of them did not exploit program properties such as highly cohesive, exclusivity of similarity and the extent of applicability to other platforms, and that was concluded from its analysis and shown in Table (3.1).

Table (3.1) Literature Review Summary

No	Author(s)	Title	Source	Technique	Result Notes	
					Strength	Limitations
1	A. Marcus, D. Poshyvanyk and R. Ferenc (2008)	Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems	IEEE Transactions on Software Engineering	Analysis of the unstructured information embedded in the source code, such as comments and identifiers which named the Conceptual Cohesion of Classes (C3), and procedure a case study on three open source software systems and compare the new measure with an extensive set of existing metrics and use them to construct models that predict software faults.	• The combination of structural and conceptual cohesion metrics gave better models for the prediction of faults in classes than of structural metrics alone. • Explained extent to impact of constructors, destructors, and assessors on increase or decrease the cohesion of a class.	• The C3 metric depends on reasonable naming conventions for identifiers and relevant comments contained in the source code. • C3 does not take into account polymorphism and inheritance in its current form.
2	G. Bavota, A. De Lucia, and R. Oliveto (2011)	Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures	Journal of Systems and Software	Graph theory was used to represent a refactored class, where each node represented a method of the class and the weight of an edge that connected two nodes (methods), and then A MaxFlow MinCut algorithm was used to split the built graph in two sub-graphs.	• The merged classes showed have cohesion much lower than the original classes. • The refactored classes have a much better quality than the merged classes.	The original class divided to two classes only.
3	P.-H. Chu, N.-L. Hsueh, H.-H. Chen, and C.-H. Liu (Mar, 2012)	A test case refactoring approach for pattern-based software development	Software Quality Journal	Illustrated test case refactoring for extreme programming which leads to faster development. Two patterns from the Gang of Four were selected for the application of refactoring and the validation of study.	This approach specified the role-based pattern structures for the functional and nonfunctional intents in the initial phase.	•The approach not an automatic completely. •Some refactorings, such as Rename, Extract Interface, or Introduce Parameter require developers to give appropriate names for the new methods, variables, or parameters.
4	J. Al Dallal (Oct, 2012)	Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics	Information and Software Technology	Logistic regression analysis was applied to predict classes in need of ESR where analysis showed a strong relation between the internal qualities attributes of a class and its need for ESR.	•A strong relation between the internal qualities attributes of a class and its need for ESR without analyzing the external relations between the class of interest and other classes. • Models constructed using the quality metrics showed high abilities to segregate classes into those that were in need and those that were not in need of refactoring.	Inspecting the whole code and analyzing the relations among the classes to identify the classes in need of ESR took time consuming.
5	M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou (Oct, 2012)	Identification and application of Extract Class refactorings in object-oriented systems	Journal of Systems and Software	An approach accompanied by a tool that worked at identifying source code chunks that collaborated to provide a specific functionality and their extraction as separate methods. Then accuracy of the proposed approach has been empirically validated both in an industrial and an open-source setting.	• Method produced meaningful and conceptually correct suggestions and extract classes. • The suggested refactorings improved the design of the system in terms of coupling.	•Suggestions could have been better and more complete if the clustering algorithm was combined with other methods, like code duplication detection techniques. •The tool interface does not provide many options for the user because it is a black box and is often difficult to understand to user.

Table (3.1) Literature Review Summary (Cont'd)

No	Author(s)	Title	Source	Technique	Result Notes	
					Strength	Limitations
6	J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian (Mar, 2014)	Functional Over-Related Classes Bad Smell Detection and Refactoring Suggestions	International Journal of Software Engineering & Applications	The detection and refactored of classes which were functionally not related where collected the dependency relationships between classes, and computed the invoking rates and compared with dynamic thresholds, then the work was validated on four open source systems- HSQLDB, Tyrant, ArgoUML and JfreeChart.	• Cohesion increased and Coupling decreased. • This approach Hierarchies, Abstraction, Encapsulation, Cohesion, Inheritance and Complexity increased; coupling decreased where increased the value of quality properties.	• The thresholds in FRC bad smell detection that are preset thresholds decrease the veracity of detection results. • Dynamic thresholds are used for bad smell detection, therefore the subjectivity is lower, and the detection is more accurate.
7	G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto (Dec,2014)	Automating extract class refactoring: an improved method and its evaluation	Empirical Software Engineering	Method chains that was used to define new classes with higher cohesion than the original class. Then a comprehensive empirical evaluation of the proposed approach performed through a two of studies. In the first study an evaluated the quality of the refactoring solutions proposed where conducted a user study with 50 graduate students asked them to rate the refactoring suggested by the proposed approach on 17 Blobs from two open-source systems. In Second study was carried out on eleven classes from six open source systems, which actually underwent extract class refactorings.	• Strongly increased the cohesion of the refactored classes without leading to a significant increase in coupling. • Bias was avoided, where students did not know the goal of this study or the techniques which produced the suggested refactoring solutions.	• Master's students represented an important threat related to the generalization of the results. As a result of difference the experience between them and professionals. • The extracted classes when splitting a Blob exhibited a worse division of responsibilities than the Blob.
8	G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba (Sep, 2015)	An experimental investigation on the innate relationship between quality and refactoring	Journal of Systems and Software	The study has been conducted on 63 releases of three open source projects, and was analyzed manually of 15,008 refactoring operations and 5478 smells. Then a search was guided by a quality evaluation function based on eleven object-oriented design metrics (i.e., the CK metrics) that accurately reflects refactoring goals and used the symbol table and reference information together with simple code metrics, such as line and statement counts.	This study has shown more often, quality standards do not show a clear relationship with housing reconstruction.	• Quality metrics did not show a clear relationship with refactoring. • The refactoring only mitigated the problem, without however necessarily removing completely the smell. • The developer's point-of-view of classes in need of refactoring did not always match with "quality indicators".
9	A. Kaur and M. Kaur (2016)	Analysis of Code Refactoring Impact on Software Quality	MATEC Web of Conferences	The approach done by used an Eclipse tool to refactor the bad smells and make a source code essay to understand. That were calculated the complexity of project and compare it with initial complexity and then check the result.	• The importance of refactoring was confirmed. • The complexity of project reduced after apply the refactoring. • Refactoring reduced the maintenance cost.	The technique was applied on a java source code only.

Table (3.1) Literature Review Summary (Cont'd)

No	Author(s)	Title	Source	Technique	Result Notes	
					Strength	Limitations
10	V. E. Zafeiris, S. H. Poulias, N. A. Diamantidis, and E. A. Giakoumakis (2017)	Automated refactoring of super-class method invocations to the Template Method design pattern	Information and Software Technology	An algorithm was used for the discovery of refactoring opportunities that is based on an extensive set of refactoring preconditions. These preconditions ensured that the suggested refactorings can be safely applied to the source code.	• A satisfactory number of Call Super was eliminated. • The Specialization Index metric (SIX) in the affected subclasses was decreased. • Runtime performance results that supported the scalability of the approach.	• The results cannot be generalized to a project or projects written in another programming language other than java. • Potential bugs in the implementation of the refactoring identification algorithm lead to underestimation of the effectiveness of this method.
11	R. Morales, F. Chicano, F. Khomh, and G. Antoniol (2018)	Efficient refactoring scheduling based on partial order reduction	Journal of Systems and Software	Automatically scheduling refactoring operations for correcting anti-patterns in software systems where conducted a case study with five open-source software systems and compared the performance of RePOR with the performance of two well-known metheuristics (GA and ACO), one conflicting-aware refactoring approach (LIU), and a recent metaheuristic based on sampling (Sway).	• RePORT can correct anti-patterns in just a fraction of the time and with less effort. • The approach relies reduction techniques from model checking.	• The approach was applied only on open source systems. • Comparing with the genetic algorithm that is known has computationally expensive, i.e. time-consuming.
12	B. Turkistani and Y. Liu (2019)	Reducing the Large Class Code Smell by Appling Design Patterns	IEEE	Classified the causes of the code smell and applied different design patterns to refactor the code to improve the quality of the software, analyzed the causes of the Large Class code smell and classified them into corresponding types and proposing a design pattern to address each type to refactor the code.	• The approach helped to refactor the code to make the maintenance, modification, and reusable easy. • Ability to use the method to reduce the long method or enhance the duplicate code, as well.	• There are different causes of the complexity, such as if, while, for, for each, case, default, continue, go to, do, and select. However, only addressed "if-else" and "switch" statements to reduce the complexity. • There are many types of cohesion like coincidental, logical, temporal, procedural, communicational, sequential, and informational. However, only addressed the logical cohesion
13	A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts (2020)	Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation	IEEE	Rejuvenate an industrial embedded software component implementing a nested state machine, where developed and applied a series of small, automated, case-specific code refactorings that ensure the code uses well-known programming idioms, then performed model-based rejuvenation focusing on the high-level structure of the code.	• Giving opportunity to be validated early in the form of code reviews and testing, since each refactoring is carried out directly on the existing code. • Aligning the code with the type of model simplifies the extraction, making the process less error-prone.	• Not specify whether refactoring after rejuvenation could be beneficial. • Focus on qualitative aspects and ignored quantitative aspects. • The time to create the refactoring steps was not considered. • The time to create the refactoring steps was not considered.

3.4 Remarks

There were a number of limitations in the relevant literature, such as application to specific systems and languages, for example Java, without indicating the possibility of generalization on other platforms and languages. Also, not taking account inheritance and polymorphism, dividing the class into two classes only and the need for developers to rename the extracted classes. This research is applied on closed system and language C#.net, divide the proposed class into two or more classes, keeping the class name, and maintain the relationship as one class between them.

3.5 Summary

In this chapter, a literature review and previous research were shown, of which, what provided ways to code smells detection and the need to address them because of their impact on software quality, and others what provided solutions to rebuild and get rid of these smells. Also, Some of these studies presented solutions to the large class smell problem, for example, the design pattern, code reorganization and cohesion metrics. The chapter also presented a summary of all previous related works, explaining the techniques and methods used, and the strengths and weaknesses of each.

CHAPTER 4: DESIGN & IMPLEMENTATION

4.1 Overview

The proposed technique for extracting the large class is designed by applying the rules of class normalization by creating a table of attributes access-set for the candidate class. Then, creating an attributes similarity matrix using the Jaccard index. Therefore, by calculating the values of the matrix the proposed classes are created. After, that the coherence of each proposed class is tested by calculating the class cohesion criterion. By testing and implementing the proposed approach of research and evaluating the results will be obtained, and in comparison with some other relevant studies, effectiveness is measured in the process of refactoring the code and extracting class. This chapter takes that in detail.

4.2 The Proposed Extract Class Approach

The proposed method for extracting large class by simulating the three rules for normalizing classes. A class is in 1ONF when specific behavior required by an attribute that is a collection of similar attributes, and when shared behavior required by more than one instance of the class is encapsulated to be 2ONF, lastly in 3ONF when it encapsulates one set of coherent behavior.

The method boils down to take a class with many responsibilities that is nominated for extracting by the developer or automatically, where the parser to produce an access-set table of attributes, then calculating the Jaccard similarity index to create a similarity matrix for attributes as shown in Figure (4.1). The structural similarity matrix is created to compute the cohesion of each class, thus achieving the third rule for normalization. In the case of a high cohesion ratio, the class is behaviorally coherent.

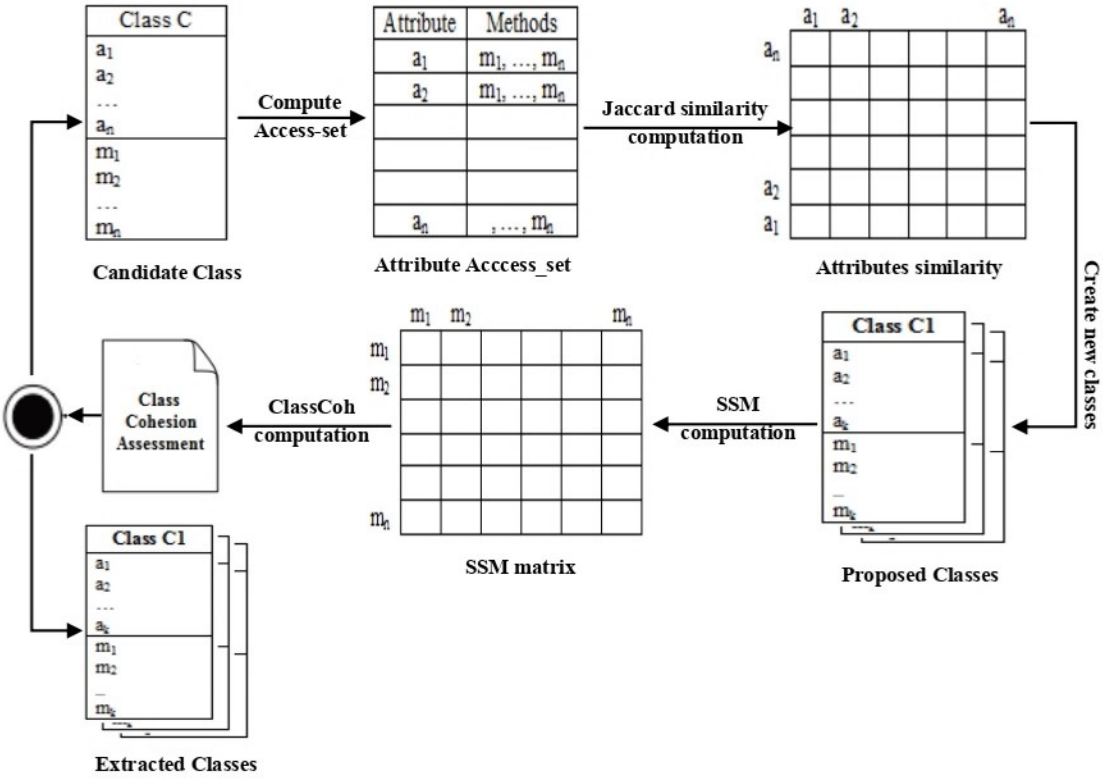


Figure (4.1) Process of Extract Class (Class Refactoring)

4.2.1 Attribute Similarity Matrix

The attributes similarity matrix is calculated by computing the Jaccard similarity ratio (Sharma & Murthy, 2014) that measures the similarity between two sample sets; it represents a ratio between the sets intersection size and the sets union size as given in Equation (4.1). Where access-set is a sample set; hence, computing Jaccard similarity between each access-sets of two attributes until form a similarity matrix for all attributes.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \qquad if\ |A \cup B| \neq 0 \qquad (4.1)$$

Similarity matrix has values in [0, 1]; where the value of 1 for a number of attributes indicates that is in the same class with the methods that related to. Consequently, a number of proposed classes are consisted as a result of the original class extraction.

4.2.2 Structural Similarity between Methods Matrix

The structural similarity matrix of constituent classes is formed using the structural similarity calculation between methods (SSM) as given in Equation (4.2) (Bavota et al., 2011):

$$SSM(mi,mj) = \begin{cases} \frac{|Ii \cap Ij|}{|Ii \cup Ij|} & if\ |Ii \cup Ij| \neq 0\ and\ i \neq j \\ 0 & otherwise. \end{cases} \quad (4.2)$$

The SSM of mi and mj is calculated as the ratio between the number of reference attributes that are shared by mi and mj methods and the total number of attributes that are referenced by both methods.

4.2.3 Compute & Assessment Class Cohesion

SSM a measure exploited to compute the cohesion metric ClassCoh, by summing the similarities of all method pairs and dividing by the total number of such pairs as given in Equation (4.3) (Gui & Scott, 2008):

$$ClassCoh = \frac{\sum_{i,j=1}^{m} SSM(mi,mj)}{m^2 - m} \quad (4.3)$$

where m is the number of class methods. According to results of calculating this metric, extracted classes are evaluated so that the value between [0.50-1.00] indicates the coherence of classes and less than that mean its incoherent and requires re-extraction.

4.3 Implementation and Test of Approach

Example: Figure (4.2) shows part of the *UserManagement* class and from its name and set of methods, this class was probably originally responsible for implementing a set of operations that would allow the user entity to be manipulated in the database. However, this class has two new responsibilities added, i.e., the Teaching Entity management and the Role Entity management. The task is to separate this class so that each entity becomes in a separate class and with a specific responsibility by defining single responsibility methods in the class. The question here, do proposed approach able that?. The names of methods in the class have been abbreviated, as follows: inserUser (IU), updateUser (UU), deleteUser (DU), existsUser (EU), checkMandatoryFieldsUser (CU), inserTeaching (IT), updateTeaching (UT), deleteTeaching (DT), checkMandatoryFieldsTeaching (CT), inserRole (IR), updateRole (UR), deleteRole (DR) and checkMandatoryFieldsRole (CR).

```
public class UserManagement
    {
        public void insertUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "INSERT INTO tblUser ...";
            ...
        }
        public void updateUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "UPDATE tblUser ...";
            ...
        }
        public void deleteUser(User pUser)
        {
            string sql = "DELETE FROM tblUser ...";
            ...
        }
        public void existsUser(User pUser)
        {
            string sql = "SELECT FROM  tblUser ...";
            ...
        }
        public bool checkMandatoryFieldsUser(User pUser)
        { ... }

        public void insertTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "INSERT INTO tblTeaching ...";
            ...
        }
        public void updateTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "UPDATE tblTeaching ...";
            ...
        }
        public void deleteTeaching(Teaching pTeaching)
        {
            string sql = "DELETE FROM tblTeaching ...";
            ...
        }
        public bool checkMandatoryFieldsTeaching(Teaching pTeaching)
        { ... }

        public void insertRole(Role pRole)
        {
            bool check = checkMandatoryFieldsRole(pRole);
            string sql = "INSERT INTO tblRole ...";
            ...
        }
        public void updateTeaching(Role pRole)
        {
            bool check = checkMandatoryFieldsTeaching(pRole);
            string sql = "UPDATE tblRole ...";
            ...
        }
        public void deleteTeaching(Role pRole)
        {
            string sql = "DELETE FROM tblRole ...";
            ...
        }
        public bool checkMandatoryFieldsRole(Role pRole)
        { ... }
    }
```

Figure (4.2) User Management Class

The following steps are representing an applying the proposed method on previous class:

Step 1. Create Access-set table for all attribute (variables) in the class as shown in Table (4.1)

Table (4.1) Attributes access-sets

Attribute	Access-set
pUser	IU, UU,DU, EU, CU
pTeaching	IT, UT, DT, CT
pRole	IR, UR, DR, CR

Step 2. Calculate the Jaccard Similarity Index of attributes to build similarity matrix as shown in Table (4.2)

Table (4.2) Attributes smilarity matrix (Jaccard)

	pUser	pTeaching	pRole
pUser	1	0	0
pTeaching	0	1	0
pRole	0	0	1

Step 3. According to the values in Table (4.2), there are three proposed classes, where each class contains attributes and methods belonging to as shown in Figure (4.3).

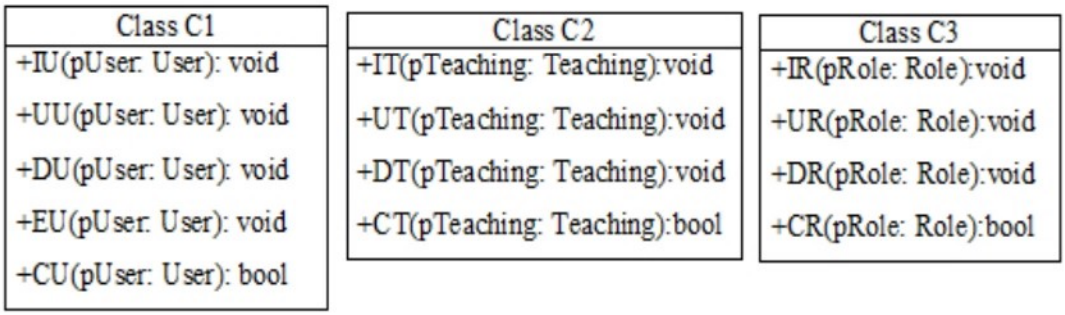


Figure (4.3) Proposed Extracted Classes

Step 4. Compute SSM for each proposed class as shown in Tables (4.3), (4.4) and (4.5)

Table (4.3) SSM Similarity of Class C1

	IU	UU	DU	EU	CU
IU		1	1	1	1
UU	1		1	1	1
DU	1	1		1	1
EU	1	1	1		1
CU	1	1	1	1	

Table (4.4) SSM Similarity of Class C2

	IT	UT	DT	CT
IT		1	1	1
UT	1		1	1
DT	1	1		1
CT	1	1	1	

Table (4.5) SSM Similarity of Class C3

	IR	UR	DR	CR
IR		1	1	1
UR	1		1	1
DR	1	1		1
CR	1	1	1	

Step 5. Compute the cohesion of each class by calculate the ClassCoh metric by using Equation (4.3).

Class cohesion of class C1, $ClassCoh = \frac{20}{25-5} = \frac{20}{20} = 1.00$

Class cohesion of class C2, $ClassCoh = \frac{12}{16-4} = \frac{12}{12} = 1.00$

Class cohesion of class C3, $ClassCoh = \frac{12}{16-4} = \frac{12}{12} = 1.00$

The results indicate each class is completely coherent. The candidate class extracted into 3 classes as shown in Figures (4.4), (4.5) and (4.6):

```
public partial class UserManagement
    {
        public void insertUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "INSERT INTO tblUser ...";
            ...
        }
        public void updateUser(User pUser)
        {
            bool check = checkMandatoryFieldsUser(pUser);
            string sql = "UPDATE tblUser ...";
            ...
        }
        public void deleteUser(User pUser)
        {
            string sql = "DELETE FROM tblUser ...";
            ...
        }
        public void existsUser(User pUser)
        {
            string sql = "SELECT FROM tblUser ...";
            ...
        }
        public bool checkMandatoryFieldsUser(User pUser)
        { ... }
    }
```

Figure (4.4) Extracted Class C1 (UserManagement)

```
public partial class UserManagement
    {
        public void insertTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "INSERT INTO tblTeaching ...";
            ...
        }
        public void updateTeaching(Teaching pTeaching)
        {
            bool check = checkMandatoryFieldsTeaching(pTeaching);
            string sql = "UPDATE tblTeaching ...";
            ...
        }
        public void deleteTeaching(Teaching pTeaching)
        {
            string sql = "DELETE FROM tblTeaching ...";
            ...
        }
        public bool checkMandatoryFieldsTeaching(Teaching pTeaching)
        { ... }
    }
```

Figure (4.5) Extracted Classe C2 (TeachingManagement)

```
public partial class UserManagement
    {
         public void insertRole(Role pRole)
         {
             bool check = checkMandatoryFieldsRole(pRole);
             string sql = "INSERT INTO tblRole ...";
             ...
         }
         public void updateTeaching(Role pRole)
         {
             bool check = checkMandatoryFieldsTeaching(pRole);
             string sql = "UPDATE tblRole ...";
             ...
         }
         public void deleteTeaching(Role pRole)
         {
             string sql = "DELETE FROM tblRole ...";
             ...
         }
         public bool checkMandatoryFieldsRole(Role pRole)
         { ... }
     }
```

Figure (4.6) Extracted Classe C3 (RoleManagement)

4.4 Evaluation of The Proposed Extract Class Approach

Based on the results of proposed approach in the preceding example. In Table (4.1), an analysis of the class elements is shown, showing each class attribute or variable and the methods belonging to. By calculating the similarity of attributes by the Jaccard metric to be shown the results in Table (4.2) and by taking the attributes that intersection between them and have a similarity value equal to 1, noticed that the pUser variable was the result of similarity with itself only and there is no relationship with other variables as well as the rest of the pTeaching and pRole attributes, so that each attribute in a class with methods belong to, Figure (4.3) shows the three proposed classes arising from the original class division. To measure the cohesion of the one

class, the calculation of the measure of the cohesion of the class is applied. So we need to calculate the structural similarity between methods and the result is appeared in Tables (4.3),(4.4) and (4.5). The results of calculating the classCoh metric showed the extent of cohesion of each class where assumed a threshold value 0.5 to be any value less than this, indicates weak the cohesive of class and needs to be refactored. Figure (4.4), (4.5) and (4.6) show the extracted classes, and the keyword partial was used to maintain class coupling, in the case of inheritance or a recall, with other classes in the system.

4.5 Comparison of Results with Previous Works

A comparison of what was achieved using the proposed approach with previous literature in obtaining extracted classes with single responsibility and more coherent elements, and with differing the used mechanisms. The approach proposed by (Bavota et al., 2011) creates a weighted graph for each class under evaluation. Class methods are treated as nodes, and cohesion is assigned as edge-weights between methods. While the presented methodology by (Fokaefs et al., 2011, 2012) that computes entity sets for each attribute and method in the target class. All entity set elements are computed with a distance matrix, and then a threshold value on distance is applied to get the cohesive sets of attributes and methods. However, considering method-calls as a primary means to establish cohesion might not well good in many cases and hence that may result in inappropriate grouping. Proposed that forming cohesive attribute-set first and then considering method-similarity as a mechanism to establish cohesion. Table (4.6) shows a comparison to the results of approaches according to specific criteria.

Table (4.6) Comparison of Approaches

Criteria	Bavota et al. Approach	Fokaefs et al Approach	Proposed Approach
Number of Extracted Class	2	>=2	>=2
Accuracy of Extracted Class Cohesion	Low	Medium	High

4.6 Summary

The chapter began by designing and explained the proposed approach for extracting large class by applying the class normalization rules (1ONF, 2ONF and 3ONF). Where the approach works to take a class with many responsibilities, then produce access-set table of attributes, and calculate the Jaccard similarity index to create attributes similarity matrix and by calculating the values of the matrix, to the proposed classes are created. The structural similarity matrix creates to compute the cohesion of each class, thus achieving the third rule for normalization, so the class is behaviorally coherent or needs re-extraction one more time. Also, it presented an example for testing the approach and evaluated according to the results obtained. And the chapter ends in the comparison of the results of the experiment for the approach in light of the results of techniques of some relevant studies for others. The method showed efficiency and ability for refactoring the large class, explaining the importance of refactoring to enhance class quality and simple the maintenance.

CHAPTER 5: CONCLUSION & FUTURE WORK

5.1 Conclusion

This study proposed an approach to extract large class and improve its cohesion. The approach splits the class to new classes with high cohesion without affecting in the coupling with other classes.

The method produces an access-set table of attributes of the class to be needed refactoring. Then, calculating the Jaccard similarity measure to create a similarity matrix for attributes and by taking the highest similarity value of intersect attributes new class are created with the methods that related to. Then, designing the structural similarity matrix of each extracted class to calculate the cohesion of each class. Class cohesion metrics, i.e. structural similarity between methods and class cohesion is applied to class normalization rules on source code.

The method shows importance of refactoring to enhance quality of class and simplest the maintenance. Where improves the structure of class and makes more organizing, and from the limitation of this increase the size of software to increase the number of classes.

5.2 Limitation of The Proposed Approach

There are some of the limitations in this research as follows:

- The application was implemented on a closed system and a specific language, C# without indicating that it can be applied to other languages such as C++ and Python.
- Using the partial keyword to keep the class as one on call and did not consider the effect of the object-oriented concepts.
- The resulting class not be named with the responsibility name which represents, but named with the original class name.

5.3 Future Work

The limitations mentioned previously can be addressed as future work. Moreover, the efficiency of the proposed method can be further improved as follows:

- Apply the proposed method to other architectures where need specific changes of the procedure of extraction refactoring according to the component's definition of the selected architectures.

- Study the effect of object-oriented concepts on class extraction refactoring.
- Enhance the approach to rename the extracted classes with names related to behaviors.

References

Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, *54*(10), 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004

Ambler, S. W. (2003). *Agile database techniques: Effective strategies for the agile software developer*. Wiley.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., & Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, *107*, 1–14. https://doi.org/10.1016/j.jss.2015.05.024

Bavota, G., De Lucia, A., Marcus, A., & Oliveto, R. (2014). Automating extract class refactoring: An improved method and its evaluation. *Empirical Software Engineering*, *19*(6), 1617–1664. https://doi.org/10.1007/s10664-013-9256-x

Bavota, G., De Lucia, A., & Oliveto, R. (2011). Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, *84*(3), 397–414. https://doi.org/10.1016/j.jss.2010.11.918

Chu, P.-H., Hsueh, N.-L., Chen, H.-H., & Liu, C.-H. (2012). A test case refactoring approach for pattern-based software development. *Software Quality Journal*, *20*(1), 43–75. https://doi.org/10.1007/s11219-011-9143-x

Dexun, J., Peijun, M., Xiaohong, S., & Tiantian, W. (2014). Functional Over-Related Classes Bad Smell Detection and Refactoring Suggestions. *International Journal of Software Engineering & Applications*, *5*(2), 29–47. https://doi.org/10.5121/ijsea.2014.5203

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, *85*(10), 2241–2260. https://doi.org/10.1016/j.jss.2012.04.013

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2011). JDeodorant: Identification and application of extract class refactorings. *Proceeding of the 33rd International Conference on Software Engineering - ICSE '11*, 1037. https://doi.org/10.1145/1985793.1985989

Fowler, M., & Beck, K. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Gui, G., & Scott, P. D. (2008). New Coupling and Cohesion Metrics for Evaluation of Software Component Reusability. *2008 The 9th International Conference for Young Computer Scientists*, 1181–1186. https://doi.org/10.1109/ICYCS.2008.270

Kaur, A., & Kaur, M. (2016). Analysis of Code Refactoring Impact on Software Quality. *MATEC Web of Conferences*, *57*, 02012. https://doi.org/10.1051/matecconf/20165702012

Marcus, A., Poshyvanyk, D., & Ferenc, R. (2008). Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, *34*(2), 287–300. https://doi.org/10.1109/TSE.2007.70768

McConnell, S. (2004). *Code complete* (2nd ed). Microsoft Press.

Mooij, A. J., Ketema, J., Klusener, S., & Schuts, M. (2020). Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 617–621. https://doi.org/10.1109/SANER48275.2020.9054823

Morales, R., Chicano, F., Khomh, F., & Antoniol, G. (2018). Efficient refactoring scheduling based on partial order reduction. *Journal of Systems and Software*, *145*, 25–51. https://doi.org/10.1016/j.jss.2018.07.076

Pressman, R. S. (2010). *Software engineering: A practitioner's approach* (7th ed). McGraw-Hill Higher Education.

Sharma, T., & Murthy, P. (2014). ESA: The exclusive-similarity algorithm for identifying extract-class refactoring candidates automatically. *Proceedings of the 7th India Software Engineering Conference on - ISEC '14*, 1–6. https://doi.org/10.1145/2590748.2590763

Singh, S., & Kaur, S. (2017). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*. https://doi.org/10.1016/j.asej.2017.03.002

Software Engineering Tutorial. (2014). Tutotorials Point (I) Pvt. Ltd.

Sommerville, I. (2016). *Software engineering* (Tenth edition). Pearson.

Turkistani, B., & Liu, Y. (2019). Reducing the Large Class Code Smell by Applying Design Patterns. *2019 IEEE International Conference on Electro Information Technology (EIT)*, 590–595. https://doi.org/10.1109/EIT.2019.8833851

Zafeiris, V. E., Poulias, S. H., Diamantidis, N. A., & Giakoumakis, E. A. (2017). Automated refactoring of super-class method invocations to the Template Method design pattern. *Information and Software Technology*, *82*, 19–35. https://doi.org/10.1016/j.infsof.2016.09.008

List of Publications

Marwan Ahmed Lardhi, Saeed Mohammed Baneamoon, Enhanced Class Normalization Rules for Refactoring Large Class Smell, **International Journal of Innovative Science and Research Technology (IJISRT)**, Vol. 5, Issue. 5, PP. 1513-1519, **2020**.