Jurnal Ilmiah Teknik Elektro Komputer dan Informatika (JITEKI)
Vol. 10, No. 4, December 2024, pp. 734-746
ISSN: 2338-3070, DOI: 10.26555/jiteki.v10i4.30277

Automatic Software Refactoring to Enhance Quality: A Review

Shahbaa I. Khaleel, Rasha Ahmed Mahmood
Department of Software, College of Computer Science and Mathematics, University of Mosul, Mosul, Iraq

ARTICLE INFO

Article history:

Received October 24, 2024
Revised December 05, 2024
Published December 13, 2024

Keywords:

Refactoring Recommendation;
Machine Learning Algorithms in Predicting Software Refactoring;
Refactoring in a Code Review;
Refactoring Prediction through Deep Learning;
Refactoring Methods

ABSTRACT

Refactoring aims to enhance the internal structure of the code and improve maintainability without affecting its functionality and external behavior. As a result of the development of technologies, it has become necessary to apply automatic refactoring to address complexities and reduce technical debt. This review presents machine learning and deep learning techniques that lead to identifying opportunities for the need for refactoring and implementing them through analyzing the software code and discovering "code smells", where the focus is on the role of tools such as RefactoringMiner, CODEBERT in enhancing the accuracy of prediction. This review presents various methodologies that include metrics-based methods, search, machine learning and discusses their impact on software quality. The review reviews experimental studies that focus on the challenges of refactoring such as reducing the risks associated with making unnecessary modifications and determining the appropriate timing. Notable empirical studies include a study by Bavota *et al*., in which Ref-Finder was used to detect 15,008 refactorings in open source software systems, identifying 85% of which improved code quality and reduced bugs. Additionally, another study by Khatchadourian *et al.* demonstrated the effectiveness of OPTIMIZE STREAMS in improving code performance in large Java projects, increasing efficiency by 55% on average. The study presents two research contributions. The first is a comprehensive analysis of automated refactoring techniques using machine learning algorithms, in addition to improving maintainability and reducing complexity. The second contribution is to provide recommendations to support developers in using modern tools and choosing the right timing for refactoring, which enhances code productivity. The results showed that machine learning techniques can significantly enhance the efficiency of refactoring and thus support developers in making accurate decisions in enhancing maintainability.

Corresponding Author:

Shahbaa I. Khaleel, Department of Software, College of Computer Science and Mathematics, Mosul University
Mosul, Iraq
Email: shahbaaibrkh@uomosul.edu.iq

1. INTRODUCTION

Software refactoring is the process of improving the internal structure of software code without affecting its external behavior in order to improve the quality and maintainability of the code. Studies show that software code deteriorates over a period of time, leading to the accumulation of what is known as "technical debt", which refers to the postponement of necessary improvements [1]. This has led to the need to apply refactoring periodically in order to maintain the quality of the code [2].

Restructuring improves software quality by increasing the speed and ease of responding to requirements. Restructuring improves software properties such as understandability, reusability, and software flexibility [3]. Refactoring has been used to identify and remove duplicate code, which leads to improving system quality [4].

Challenges of implementing refactoring when dealing with large databases, when there is a high dependency between components, and the need for developers to coordinate with other teams [5]. Developers

face challenges in identifying the right moments to refactor, as well as the risks that can occur when making unnecessary changes. When a developer manually searches or relies on experience to identify parts that require refactoring, identifies techniques, implements them, and then measures their impact on the program, the program may produce incorrect results [6].Research has shown that using machine learning techniques to identify refactoring opportunities has yielded more accurate results, helping to reduce reliance on personal intuition and individual experience [7].

Most projects relied on the use of the Refactoring Miner tool (Tsantalis et al., 2018) and it gave highly efficient results in finding refactoring operations compared to other tools. This tool provides the ability to operate using the application programming interface, so it is more suitable for applications [8].

With the development of deep learning techniques, many attempts have emerged to use these techniques to support restructuring processes. These techniques focus on two categories where they are used to detect code smells, which indicate that there is a problem in the program and recommend implementing restructuring processes [9]. Researchers have proposed many techniques to support refactoring processes by identifying strategies such as code smell detection strategies. It was found that the most used techniques for detecting code smells are search-based techniques (30.1%), symptom-based techniques (19.3%), metrics-based techniques (24.1%) [10], and logical programming [11].

The refactoring process involves a set of steps designed to improve and effectively measure the quality of software code. The goal of these steps is to improve the structure of the code and make it more understandable and maintainable, which contributes to enhancing the overall performance of the program and helps reduce potential future complexity [12]. Fig. 1 explains the steps and activities involved in the refactoring process. First, the modules will be tested to ensure that the program works as expected without errors. The code is examined to identify problems based on indicators known as code smells. Based on the problem, an appropriate refactoring method is determined to solve the problem. The refactoring method is applied and then tests are conducted to verify that the original functionality of the program is not affected. After that, the quality of the code is measured after making the modifications using software quality standards [13].

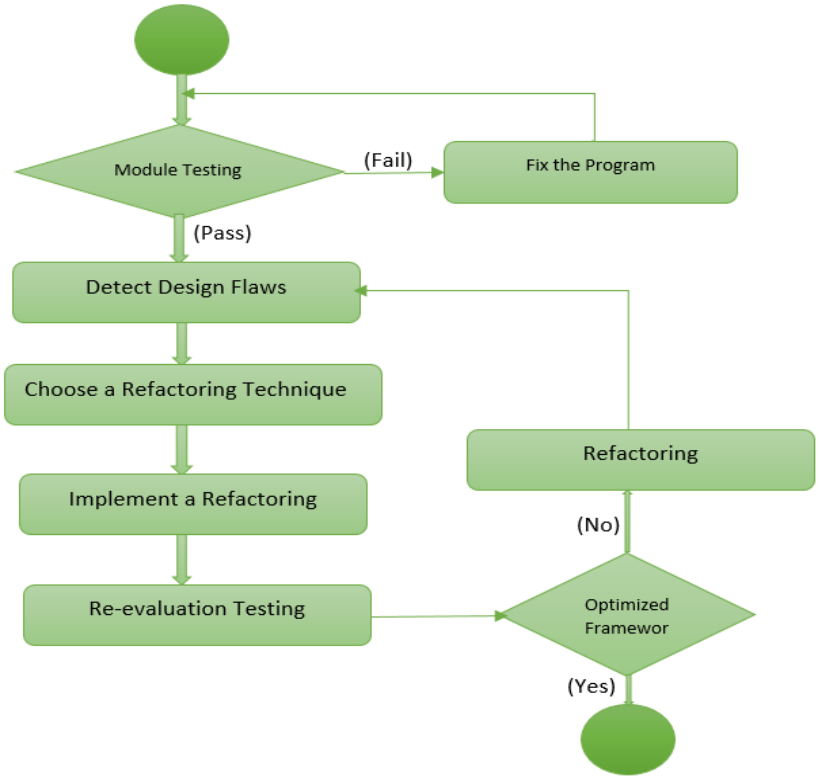

Fig. 1. The Step in Refactoring Process

The research was organized as follows: The second part addressed the clarification of restructuring recommendations that aim to identify appropriate opportunities for implementing restructuring in addition to identifying the best ways to improve the code. The third part addressed machine learning and its important role in predicting restructuring. The fourth part addressed code reviews that help identify problems and changes in the code and determine the extent of the need to implement restructuring methods. The fifth part addressed

deep learning and its use in predicting restructuring. The next part addressed restructuring methods, which explains some of the widely used restructuring methods. The seventh part addressed previous studies and researchers' results in predicting restructuring . The last part addressed the conclusions reached by the research by presenting the work of researchers in this field.

2. REFACTORING RECOMMENDATION

The goal of refactoring recommendations is to solve two main problems in the field of automated software refactoring. The first problem is to identify refactoring opportunities, which is to identify places in the code that can be improved. The second problem is to choose the right type of refactoring, which is to choose the optimal method for improving the code [14]. Researchers have proposed several methods for making refactoring recommendations, and these methods fall into three main categories:

2.1. Metrics-Based and Rule-Based Approaches

Quantitative metrics are commonly used to evaluate the design quality of software systems. However, these metrics may be insufficient when used alone to measure all aspects of the design. Therefore, an approach known as metrics-based detection strategy has been developed to detect deviations from good design principles. Using detection strategies, a developer can identify classes affected by a defect (e.g., God class, God method) [15]. A framework called DECOR has been proposed that represents a comprehensive methodology for identifying code and design defects. The methodology is based on steps to analyze the discovered defects and provide possible solutions. DETEX is a practical application of the DECOR methodology, enabling developers to identify defects at a high level of abstraction and automatically convert them into executable detection algorithms. DETEX has proven its efficiency when implemented on open source systems such as XERCES v2.7.0, where it resulted in a detection accuracy rate of 60.5% and a full recall 100% [16].

2.2. Search-Based Approaches

Search-based software engineering (SBSE) is an important approach in the field of software optimization. Different heuristics such as genetic algorithms and ant colony optimization have been used to address complex problems such as refactoring software [17]. Researchers have proposed using Pareto optimization, which simplifies metric integration and provides users with multiple optimal refactoring options [18]. SBSE has been shown to be highly efficient in dealing with large and complex codes, and also balances multiple goals such as reducing complexity and improving performance [19]. In the process of software refactoring through search, solutions are represented in several ways such as the program itself, an abstract structure tree (AST), or an abstract model [20].

2.3. Machine Learning Approaches

Machine learning can be used to analyze big data about software projects and identify which parts of the code need to be refactored the most based on quality and performance criteria. The FITTED framework is a methodology for evaluating the positivity or negativity of implementing refactoring techniques by comparing the current code to the optimal code, and it evaluates the quality of object-oriented classes based on their proximity to the optimal quality criteria. Data on refactorings was collected using the "RefactoringMiner tool. Based on this data,". a machine learning model was created with about 70% accuracy to predict when refactoring is needed in order to reduce technical debt [21].

Refactoring is a process that involves improving the design of a program by making transformations in the code without compromising its basic functionality. These transformations lead to improving the quality and efficiency of the code. Fig. 2. shows the activities involved in this process, which are illustrated by steps that show the implementation stages. This process depends on detecting code smells, which in turn indicate inefficient designs such as repetition, ambiguity, and inconsistency. Therefore, transformations are used to analyze and detect smells [22].

3. MACHINE LEARNING ALGORITHMS IN PREDICTING SOFTWARE REFACTORING

Machine learning is used to predict refactoring through predictive models trained on data from previously refactored software projects. Six different machine learning algorithms are trained on data including more than two million refactorings from 11,149 software projects. The results showed that the random forest algorithm was more accurate in predicting refactorings, with an accuracy rate often exceeding 90% [23].

Recent studies have proven the effectiveness of applying the SVM algorithm in predicting the code areas that require restructuring. The reason behind using this algorithm is its ability to perform data classification with high accuracy by forming an ideal linear interval within a multidimensional space, so it is effective in identifying the parts that require restructuring. The SVM algorithm contributes to enhancing the prediction

performance compared to other algorithms such as Naïve Bayes and Random Tree, where the prediction accuracy ranged between 84% and 93% and the F-measure accuracy ranged between 86% and 96% when applying optimization algorithms [24].

In Ratzinger et al.'s study, classification algorithms J48, LMT, Rip, and NNge were used to predict the appropriate areas for refactoring based on the project's development history [25]. Kumar and Sureka proposed an automated tool that aims to help developers identify the classes that need refactoring by using machine learning techniques such as LSSVM, PCA, and SMOTE [26].

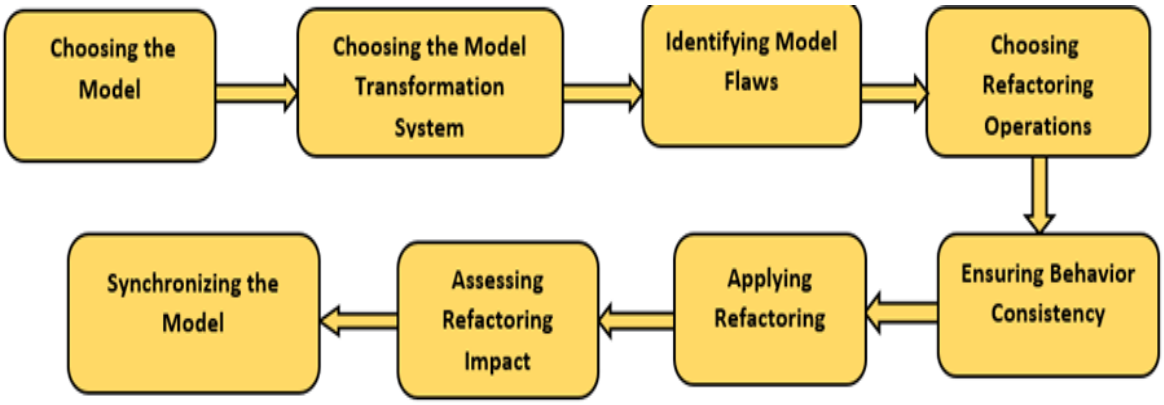

Fig. 2. Model refactoring process

4. REFACTORING IN A CODE REVIEW

Code reviews promote knowledge sharing among team members. These reviews help to improve code clarity. By using reviews, members can offer new and useful ideas. Refactoring allows for the immediate implementation of proposed ideas, making it easier to see the results of improvements and contributing to more organized and clear code [27]. Developers review code to identify changes that cause errors in the code. The refactoring-aware tool was built to identify changes that contain refactoring and differentiate them from other changes that may contain errors [28].

While reviewing the code and discovering the presence of code smells that indicate complexity and excessive duplication, the developer works on applying refactoring techniques such as Extract Method, Rename Variables, Reduce Duplication. The goal of this application is to improve readability and remove duplication. The team works on manually reviewing the changes to ensure that the new code achieves the required goals. In addition, analysis tools such as SonarQube were used to analyze the impact of the changes [29].

5. REFACTORING PREDICTION THROUGH DEEP LEARNING

Deep learning techniques in software engineering have contributed to solving many complex programming problems [30]. Deep learning (DL) techniques play a vital role in the feature discovery (FE) process [31]. CODEBERT is a pre-trained model that combines natural language and programming languages, which improves applications for example: It helps in searching for code to identify the appropriate code based on inputs written in Arabic, It leads to generating accurate text documents for code functions, CODEBERT was chosen because it provides an efficient conversion of code into vectors that can be applied to models, which leads to an easy training process [32].

Recurrent neural networks (RNN), convolutional neural networks (CNN), and multi-layer perceptrons (MLP) are the most popular in the field of refactoring, with MLP proving to be the most promising in terms of performance[33] .

6. REFACTORING METHODS

Below is a brief explanation of each of the ten common restructuring methods that were selected [34], [35].

Add Parameters (AP): When the method in programming requires additional data, this technique will simply be implemented, and a new element will be provided that passes the required data.

Encapsulate Fields (EF): This technique is used to change the access rights to data. Access is transferred from public fields to private fields.

Extract Class (EC): When the current class is too large and contains multiple tasks, a new class is created, and the tasks and related fields are moved to the new class.

Extract Methods (EM): Phrases are extracted from an unintelligible method and the phrases are assembled into a new method that is created.
Hide Methods (HM): When a method is not used by other classes or is implemented only within the class hierarchy, the method is protected by implementing this technique.
Inline Classes (IC): When a class has no future role, all methods are copied to another class and the unnecessary class is removed.
Extract Interface (EI): When there is a section of the class interface that is used by many users, this technique will be used.
Push Down Fields (PDF): If a field is applied in only some of the subcategories, this technique moves that field from the supercategory to the relevant subcategories.
Remove Parameters (RP): If the method does not need a particular parameter, that parameter is omitted.
Rename Methods (MM): If a method does not perform its function clearly, the technique is used to modify the method name to something that better expresses its function.

Table 1 shows the use of seven case studies (BMS, PMS, JGraphX, JHotDraw, Xerces, and jEdit) and the collection of some object-oriented metrics (DSC, ANA, DAM, DCC, CAM, MOA, NOP, CIS, NOM) used to measure the quality of internal features such as (abstraction, encapsulation, coupling, cohesion, composition, inheritance, and complexity), and refactoring techniques (EM, MM, EI, PDF) were applied through the case studies. Each refactoring technique was applied to determine its impact on internal quality.

Table 1. The table shows the refactoring methods (EM, MM, EI, PDF) that affect object-oriented metrics in the studies described below

Case study	RT	TA	DSC			CIS			MOA			DAM			NOP			CAM			DCC			ANA			NOM		
			↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓
BMS	EM	41	0	41	0	41	0	0	0	41	0	0	41	0	0	41	0	0	4	37	0	41	0	0	41	0	41	0	0
	PDF	3	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0
	EI	4	0	0	0	0	4	0	0	4	4	0	0	0	0	4	0	4	0	0	4	0	0	0	0	4	0	0	0
PMS	EM	4	0	4	0	4	0	0	0	4	0	0	4	0	0	4	0	0	0	4	0	4	0	0	4	0	4	0	0
	MM	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	1	0	0
	PDM	2	0	2	0	2	0	0	0	2	0	0	2	0	0	2	0	0	0	2	0	2	0	0	2	0	2	0	0
	EI	2	2	0	0	2	0	0	0	2	0	0	2	0	0	2	0	2	0	0	0	2	0	0	0	2	2	0	0
Jhot DRAW	EM	51	0	51	0	17	34	0	0	51	0	0	51	0	0	51	0	20	11	20	2	49	0	0	51	0	45	6	0
	MM	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	1	0	0
	PDM	4	0	4	0	3	0	1	0	4	0	0	4	0	0	2	2	4	0	0	0	4	0	0	4	0	2	0	2
	EI	10	10	0	0	10	0	0	0	10	0	0	10	0	0	10	0	10	0	0	5	5	0	0	0	10	10	0	0
JGRAPHX	EM	42	0	42	0	42	0	0	0	42	0	0	42	0	0	42	0	23	0	19	0	42	0	0	42	0	35	7	0
	MM	15	0	15	0	12	3	0	0	15	0	0	15	0	0	15	0	10	1	4	1	14	0	0	15	0	12	3	0
	EI	13	13	0	0	12	1	0	0	13	0	0	13	0	0	13	0	9	4	0	2	11	0	0	0	13	13	0	0
XERCES	EM	61	0	61	0	61	0	0	0	61	0	0	61	0	0	61	0	16	10	35	3	58	0	0	61	0	54	7	0
	MM	31	0	31	0	31	0	0	0	31	0	0	31	0	0	30	1	18	1	12	7	21	3	0	31	0	21	10	0
	EI	12	12	0	0	11	1	0	0	12	0	0	12	0	0	12	0	7	5	0	4	8	0	0	0	12	12	0	0
JEDIT	EM	67	0	67	0	20	47	0	0	67	0	0	67	0	0	67	0	19	23	25	5	62	0	0	67	0	48	19	0
	MM	52	0	52	0	36	16	0	0	52	0	0	52	0	0	52	0	32	0	20	11	41	0	0	52	0	37	15	0
	PDF	1	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
	EI	10	10	0	0	10	0	0	0	10	0	0	10	0	0	10	0	10	0	0	1	9	0	0	0	0	10	0	0

TA refers to the total number of times the refactoring techniques were implemented. The symbol (↑) It indicates that the restructuring technique has led to an improvement in the quality attribute. The symbol (↓) it indicates that the restructuring technique has led to a weakening of the quality attribute. The symbol (-) it indicates that the restructuring technique did not lead to any change in quality.

Table 2 shows the use of seven case studies (LMS, BMS, PMS, JGraphX, JHotDraw, Xerces, and jEdit) and a set of metrics (reusability, flexibility, effectiveness, scalability, functionality, and understandability) used to measure external feature quality, and the refactoring techniques (RP, IC, AP, EF, EC, and HM), and the Total Quality Index (TQI) were applied across the case studies. Each refactoring technique was applied to determine its impact on external quality.

TA refers to the total number of times the refactoring techniques were implemented. The symbol (↑) It indicates that the restructuring technique has led to an improvement in the quality attribute. The symbol (↓) it indicates that the restructuring technique has led to a weakening of the quality attribute. The symbol (-) it indicates that the restructuring technique did not lead to any change in quality.

Table 2. A summary of every refactoring methods (RP, IC, AP, EF, EC, HM) effect on external quality characteristics and TQI

Case study	RM	TA	Reusability			Flexibility			Effectiveness			Extendibility			Functionality			Understandability			TQI		
			↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓	↑	-	↓
LMS	EF	2	0	0	2	0	0	2	0	0	0	2	0	2	0	0	0	0	2	2	0	0	2
	EC	3	3	0	0	3	0	0	3	0	0	0	0	3	3	0	0	0	0	3	3	0	0
	HM	2	0	0	2	0	2	0	0	2	0	0	2	0	0	0	2	0	2	0	0	0	2
	IC	1	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1
BMS	AP	1	1	0	0	0	1	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	0
	EC	7	7	0	0	7	0	0	7	0	0	0	0	7	7	0	0	0	0	7	7	0	0
	HM	32	0	0	32	0	32	0	0	32	0	0	32	0	0	0	32	0	32	0	0	0	32
PMS	AP	1	1	0	0	0	1	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	0
	EF	20	20	0	0	2	18	0	2	18	0	0	20	0	20	0	0	0	18	2	20	0	0
	EC	2	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	0	0	2	2	0	0
	HM	23	0	0	23	0	23	0	0	23	0	0	23	0	0	0	23	0	23	0	0	0	23
	IC	2	0	0	2	0	0	2	0	0	2	2	0	0	0	0	2	2	0	0	0	0	2
	RP	2	0	0	2	0	2	0	0	2	0	0	2	0	0	0	2	0	0	2	0	0	2
jHOT Draw	AP	16	0	16	0	0	16	0	0	16	0	0	16	0	0	16	0	0	16	0	0	16	0
	EF	39	39	0	0	7	32	0	7	32	0	0	39	0	39	0	0	0	28	11	39	0	0
	EC	13	13	0	0	13	0	0	13	0	0	0	0	13	13	0	0	0	0	13	13	0	0
	HM	164	0	0	164	0	164	0	0	164	0	0	164	0	0	0	164	0	164	0	0	0	164
	IC	9	0	0	09	0	0	9	0	0	9	9	0	0	0	0	9	9	0	0	0	0	9
	RP	10	0	10	0	0	10	0	0	10	0	0	10	0	0	10	0	0	10	0	0	10	0
jEdit	AP	34	0	34	0	0	34	0	0	34	0	0	34	0	0	34	0	0	34	0	0	34	0
	EF	104	104	0	0	73	31	0	73	31	0	0	104	0	104	0	0	0	20	84	104	0	0
	EC	77	77	0	0	77	0	0	77	0	0	0	0	77	77	0	0	0	0	77	77	0	0
	HM	230	0	0	230	0	230	0	0	230	0	0	230	0	0	0	230	0	230	0	0	0	230
	IC	40	0	2	38	0	0	40	0	0	40	0	40	0	0	0	40	40	0	0	10	0	30
	RP	14	0	14	0	0	14	0	0	14	0	0	14	0	0	14	0	0	14	0	0	14	0

7. PREVIOUS STUDIES

There are several problems that appear while using code refactoring tools, such as slow performance, unclear messages displayed by the tools when an error occurs. This leads to programmers being slow and hesitant to use these tools efficiently. Researchers Hill and Black suggested using three new tools that were developed to improve the code refactoring process, which are Selection Assist, Box View, and Refactoring Annotations. The results showed that these tools helped improve the programmers' experience during code restructuring, as they increased the speed of performance in the code restructuring process, and reduced the percentage of errors that occur during the process. Some metrics used to measure the performance quality of the three tools: Total correctly identified phrases: 355, Total incorrectly identified phrases: 6, Average selection time: 5.5 seconds These values represent the quality of the Selection Assist tool, Total correctly identified phrases: 357, Total incorrectly identified phrases: 2, Average selection time: 7.8 seconds These values represent the quality of the Box View tool, For the Refactoring Annotations tool, Average time to identify all violated preconditions: 46 seconds, Missed violations: 1 [36].

When there are long and complex codes present in programming projects, and they include multiple parts that are not logically connected to each other, this will lead to interference between modules and reduce the coherence of the code. Researchers Tsantalis and Chatzigeorgiou proposed using a refactoring approach called "method extraction refactoring opportunities" to solve the problem. This methodology aims to refactor the code through two methods. The first method deals with variables whose values are changed by assignment instructions within the original method. The second method focuses on object references whose state is affected by method calls within the original method. Experiments have shown that using the proposed approach has led to a significant improvement in code coherence. Some metrics to measure the performance of the proposed approach. Precision, recall and coherence metrics were used [37].

To see how refactoring affects code quality, Bavota *et al.* used the Ref-Finder tool to detect refactorings in 63 versions of three open source Java software systems (Apache Ant, Xerces-J, ArgoUML). Using Ref-Finder, they detected 15,008 refactorings, which were manually checked and found that 12,922 of them were correct. They then applied the SZZ algorithm to determine whether these refactorings resulted in bugs. The results showed that 15% of the detected refactorings were associated with subsequent bug fixes. The Precision metric was used to measure the quality of the Ref-Finder tool, which was 79%, and the Recall metric was also used to measure the quality of the tool, which was 95% [38].

Over time, the system structure deteriorates and its quality decreases, a phenomenon known as software decay. To address this problem, researchers El-Boussaidi and Ghannem presented an approach that aims to identify appropriate sequences of refactorings. An interactive genetic algorithm (IGA) was adapted, which aims to improve the quality of the software model design through an evolutionary process that combines the

knowledge gained from previous refactoring examples and the feedback provided by designers during the optimization process. This algorithm contributed significantly to reducing the number of useless refactorings in optimal solutions. The measure used to measure the quality of the algorithm is the Fitness Function. In addition, the Precision and Recall measures were used. The percentage for the Precision measure was about 90%, while the Recall measure was 88% [39].

During software development, systems will undergo modifications that may complicate the source code, leading to a decrease in quality, especially in class cohesion and increased coupling between them. Therefore, researchers Bavota *et al.* proposed an automated method for class refactoring using the Extract Class Refactoring technique. This method is based on analyzing the relationships between functions within a class, and the goal is to identify closely related function chains. Based on these chains, new classes are created that have higher cohesion and maintain the coupling ratio. The LCOM (Lack of Cohesion in Methods) metric was used to measure the quality of structural cohesion, while the C3 metric was used to measure semantic cohesion [40].

Since software maintenance is the most expensive phase in the software development life cycle, due to frequent modifications, the code quality deteriorates. Malhotra and Chugh proposed a set of refactoring techniques including (conditional expression unification, field encapsulation, extraction method, class extraction, and hiding method). The techniques were applied to five different software systems, which provided a complete view of the impact of refactoring. It was found that refactoring significantly improves the quality of software and improves its maintainability. The quality of these techniques was measured using internal and external metrics. The internal metrics used are WMC, DIT, NOC, CBO, RFC, and LCOM. While the internal metrics used are understandability, abstraction level,modifiability, extensibility, and reusability [41].

Improving software quality requires continuous changes, and the challenge is to make these changes automatically while maintaining code quality. To solve this problem, researchers Mohan and Greer introduced MultiRefactor, a tool that improves quality by automating code refactoring. The tool uses search-based optimization techniques, including multi-objective genetic algorithms and single-objective algorithms, to guide optimization processes. The tool refactors Java projects using these algorithms while ensuring that the code is correct. MultiRefactor also has the ability to handle multi-objective optimization processes to speed up maintenance. Some metrics for assessing software quality after refactoring include coherence between methods and visibility ratio [42].

The problem revolves around the inflexibility of current networks, especially LTE (Long Term Evolution) networks, which are designed to provide specific services to mobile operators. These networks rely on dedicated implementation devices for specific individual functions. The negative consequences are signaling storms that occur when unnecessary signals are exchanged within the network due to frequent or intensive use of some applications. Pozza *et al.* proposed Network Functions Restructuring. which involves an approach to segment network functions in LTE networks into smaller, more specialized units that can interact with each other flexibly. This is achieved by using technologies such as Software-Defined Networking (SDN), where the control and management layers are separated from the data plane and control plane. This technology aims to reduce the load on the network and enhance its performance, reducing "signaling storms" that may cause performance degradation. The number of signals exchanged between network functions has been used as a measure of the quality of Network Functions Restructuring [43].

The problem revolves around the difficulty of accessing websites for people with disabilities. There are Web Content Guidelines (WCAG) that aim to enhance accessibility, but most websites do not apply these standards effectively. Because correcting codes to make websites comply with these standards takes a lot of time and effort. Researchers Ikhsan and Candra introduced the Automated Refactoring technique, which is a process that automatically corrects codes to enhance accessibility according to WCAG standards. To achieve this, a tool called Automatica11y was developed that detects problems in codes and automatically applies techniques from WCAG to fix them. The results showed that applying the Automatica11y tool proved effective by reducing the number of errors in codes faster than the manual correction process. This led to improving the experience of users with disabilities in interacting with these websites. The tool was evaluated by comparing the number of errors and warnings before and after using the tool [44].

To enhance the performance of using the Java Streams API, Java 8 Streams allows operations on data to be performed in MapReduce mode, making it easier to write parallel programs. However, effective use of this API requires careful considerations such as determining when operations should be performed in parallel, and when these operations might be less efficient when performed in parallel. To solve the problem, Khatchadourian and others proposed an open source tool called OPTIMIZE STREAMS, which automatically reworks Java 8 streams to make them more efficient. The tool is based on a new analysis called ordering and type case analysis, which helps improve stream performance by determining when data order in expressions should be maintained. Type case analysis is used to safely improve stream performance when running in

parallel or serially. The tool has helped developers significantly improve their code. The Speedup metric was used to measure the tool's performance. An average speedup of 1.55 was recorded, meaning that programs became 55% faster on average after implementing OPTIMIZE STREAM [45].

Mobile applications, especially those running on Android. The problem revolves around the fact that applications consume a lot of battery power due to energy-inefficient development practices by developers. To solve this problem, researchers Cruz and Abreu proposed applying automatic refactoring strategies to improve the energy efficiency of applications. A tool called Leafactor was developed that automatically refactors the application code to reduce energy-inefficient codes (energy code smells) that developers usually ignore. The results showed that using the automatic refactoring tool significantly helped in improving the energy consumption of applications. The energy efficiency optimization metric and lint priority were used to measure the performance of the proposed approach [46].

The main problem is how to improve the flexibility and maintainability of high-quality software automatically by identifying code smells and refactoring them without the need for manual intervention from developers. To solve the problem, Nassagh *et al.* proposed an approach based on Fuzzy Genetic Algorithm (FGA) using a graph model to automatically find and refactor code smells. The code is represented as a graph model. The algorithm applies a fuzzy rule generated by using genetic algorithmes to identify code smells and apply refactoring. The results showed the effectiveness of this approach compared to other traditional methods. Some quality metrics were used to measure the effectiveness of the method. The most prominent of these metrics include: betweenness centrality, load centrality, and closeness centrality [47].

The difficulty in maintaining object-oriented software is the accumulation of "bad smells" in the source code, which leads to deeper design problems and increased costs. To solve the problem, researchers Tarwani and Shog proposed using refactoring techniques to remove bad smells from the source code. The most effective hill climbing algorithm was used to determine the optimal sequence of refactoring techniques. The proposed algorithm enables maintenance teams to improve software quality effectively and in less time, making it an important and effective tool in software maintenance. To measure the performance of the proposed method, the following were used: the simple hill climbing algorithm, the most difficult hill climbing algorithm, and the random hill climbing algorithm [48].

The difficulty in using the Extract Method tool correctly is due to the poor choice of code by the developers. The Extract Method faces significant usability issues. Researchers found that this tool has a failure rate of up to 49% when developers try to use it. Therefore, Alcocer *et al.* proposed designing a new tool called TOAD (Tool for Alternative Refactoring Design). TOAD aims to improve the success rate of applying the Extract Method refactoring by suggesting alternative choices for the chosen code. The tool presents five alternative choices to the user's original choice, making it easier to find the correct part of the code that can be successfully refactored. This tool has proven to be effective in improving the success rate when using the Extract Method. The Success Rate metric was used to measure the quality of the tool [49].

The problem revolves around improving software models using multi-objective optimization techniques. These models face problems in improving non-functional aspects such as performance and reliability, which typically require significant time and resources. Researchers Pompeo and Tucci used a technique which is the application of multi-objective genetic algorithms such as NSGA-II, SPEA2, and PESA2 to perform optimizations on software models under a specific time constraint known as the search budget. These algorithms automatically generate design alternatives and evaluate them based on a set of indicators such as performance and reliability. The results showed that reducing the search budget significantly reduces the quality of the generated solutions. It was also shown that the type of algorithm used plays a crucial role in the quality of the results. Hypervolume (HV) is a good measure for evaluating the performance of algorithms in multi-objective optimization [50].

Optimizing business processes designed using BPMN (Business Process Modeling) language. Optimizing these processes has become a strategic issue for companies in order to reduce costs and increase efficiency. The main problem is that manual process optimization is complex due to the factors that must be taken into account such as execution times and resources used. Process optimization also requires a high degree of expertise. Researchers Duran and Sallon proposed a technique based on process reengineering or what is known as "automated reengineering". This is done by reorganizing tasks and taking into account the resources used by these tasks. The goal is to increase the parallelism between tasks so that the total execution time is reduced. These steps are performed automatically using a tool developed for this purpose. The proposed technique has been applied to several examples and tangible results have been achieved in improving efficiency and reducing execution time. The metric used to measure the quality of (automated reengineering) is the average execution time (AET) [51].

When software quality is optimized for performance and reliability, improving these attributes can negatively affect other attributes. The challenge is to determine the equivalence between these attributes. To

solve the problem, researchers Cortellessaa *et al.* proposed a genetic algorithm called NSGA-II (Non-Dominated Genetic Sorting Algorithm II) to analyze the optimum and generate Pareto optimal frontiers for software models after refactoring. The technique is based on the use of UML models augmented with MARTE and DAM profiles that contain performance and reliability characteristics. The results showed that multi-objective optimization techniques are effective in balancing performance and reliability in software models, while maintaining the quality of the resulting solutions. The metrics used to measure the quality of NSGA-II include: performance quality index (perfQ), reliability, performance anti-patterns, and architectural distance. These metrics were used to evaluate the solutions generated using NSGA-II [52].

Table 3. Shows A Summary Of Previous Studies of Refactoring.

Author	Methodology	Result
Hill and Black [2008]	Selection Assist, Box View, Refactoring Annotations	Selection Assist reduced the average selection time from 10.2 seconds to 5.5 seconds, and Box View reduced the selection time to 7.8 seconds. Error rate is lower with Refactoring Annotations.
Tsantalis and Chatzigeorgiou [2011]	Extract Method refactoring opportunities	Improve maintainability, Improve internal code consistency, Reduce code duplication.
Bavota, *et al.* [2012]	Ref-Finder	The result of applying the Ref-Finder tool is to identify refactoring operations in software system, some of which were found to contribute to correcting errors, such as "pull method" and "subclass extraction", by up to 40%. While the percentage of errors resulting from refactoring was generally low (15%).
Ghannem and El-Boussaidi [2013]	(Interactive Genetic Algorithm - IGA)	The results showed that the precision and recall were about 88% on average, demonstrating that the proposed approach generates refactoring sequences that often match those actually applied.
Bavota, *et al.* [2013]	Extract Class Refactoring	It greatly increases the coherence of extracted classes without significantly increasing coherence. It is useful for engineers who refactor code. It is able to approximate solutions implemented manually by developers by up to 91% on average.
Chug and Malhotra [2016]	Consolidate Conditional Expressions, ,Encapsulate Fields , Extract Methods Extract Classes, Hide Methods	CCE: Increases maintainability in terms of internal and external attributes of quality. EF: Reduces maintainability but makes the class safer by converting public members to private. EM: Makes classes more clear and organized, increasing maintainability. EC: Reduces code quality and maintainability due to high coupling between classes, but improves reusability. HM: Does not change internal attributes of quality but slightly improves external attributes, which enhances maintainability.
Mohan and Greer [2017]	MultiRefactor	Single-objective search techniques: gave better results when only one metric was optimized. Multi-objective search techniques: gave generally good results while saving time.Multi-objective optimizations took 71% less time than applying all single-objective search techniques combined.
Pozza *et al.* [2017]	(Refactoring Network Functions) with (Software-Defined Networking - SDN).	As a result of applying this technology, correct network restructuring can significantly reduce the number of signals exchanged between different units in the network, thus reducing the load on the network and improving performance.
Ikhsan and Candra [2018]	Automatica11y	Error: Ref-1 (15), Ref-2 (5) Warning: Ref-1 (9), Ref-2 (9)
Khatchadourian *et al.* [2018]	OPTIMIZE STREAMS	The tool was evaluated on 11 Java projects containing approximately 642 thousand lines of code. The results showed that : %36.31of the nominated streams were reformattable ,The average speedup in performance was 1.55 times.
Cruz and Abreu [2018]	Leafactor	This technique proves useful in improving the power efficiency of mobile applications without requiring significant manual intervention by developers, reducing the likelihood of errors and increasing the efficiency of the development process.

Author	Methodology	Result
Nasagh *et al.* [2020]	fuzzy genetic algorithm	The proposed approach can identify 68.92% of the defective code classes and refactor 77% of them successfully.
Tarwani and Chug [2020]	Steepest-Ascent Hill-Climbing	The results showed that applying the Steepest-Ascent Hill-Climbing algorithm improved the maintainability value by 20%, which means a significant improvement in software quality.
Alcocera *et al.* [2020]	TOAD (Tool for Alternative refactoring Design)	After conducting a comparative experiment between TOAD and a standard refactoring tool in the Pharo environment, the researchers found that TOAD was able to significantly reduce the number of failed refactoring attempts.
Pompeo and Tucci [2022]	NSGA-II, SPEA2, PESA2	The NSGA-II algorithm was the fastest at finding solutions, while PESA2 was the most capable of producing high-quality solutions but took longer. As for the SPEA2 algorithm, it was the slowest and least efficient in producing high-quality solutions.
Durán,and Salaün [2022]	Automated Refactoring	The results showed that the technology is able to improve operations by reducing the total execution time, and increasing efficiency by organizing tasks in a way that ensures the use of available resources.
Cortellessaa *et al.* [2023]	NSGA-II (Non-dominated Sorting Genetic Algorithm II)	Results showed up to 42% improvement in performance while maintaining or improving reliability by up to 32%.

8. CONCLUSION

Refactoring is important in improving software quality and facilitating its maintenance, and it is a common practice due to the evolution of technology and the need for software with multiple and changing requirements over time. This study focused on how to implement and use refactoring that enhances its internal structure, while maintaining external performance, which leads to reducing technical debt and improving performance. It also discussed multiple approaches and methods for refactoring, such as those based on metrics and rules, in addition to the use of search and machine learning techniques in improving software design as well as the tools and techniques used in the refactoring process, and the challenges associated with it, and showed the role played by machine learning and deep learning in discovering optimal refactoring opportunities, for example, tools such as RefactoringMiner and CODEBERT have shown a significant role in enhancing the accuracy of refactoring prediction. It also discussed the importance and role of code reviews in improving its accuracy and knowledge sharing among team members. Some of the results of previous studies were presented, showing that techniques such as NSGA-II and tools such as OPTIMIZE STREAMS lead to performance enhancement while reducing structural complexity, which illustrates the impact of refactoring on software quality and maintainability. The results provide practical insights for developers, as automatic refactoring techniques can reduce maintenance effort. However, the ethical impact of these tools should be considered, such as verifying the accuracy of the refactored code and its impact on software functionality.

Acknowledgments

Authors would like to thank the University of Mosul in Iraq for providing Moral support.

REFERENCES

[1] E. Zabardast, J. Gonzalez-Huerta, and D. Smite, "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 376–384, Aug. 2020, https://doi.org/10.1109/SEAA51224.2020.00068.

[2] M. Tufano *et al.*, "When and Why Your Code Starts to Smell Bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 403–414, May 2015, https://doi.org/10.1109/ICSE.2015.59.

[3] H. Khosravi and A. Rasoolzadegan, "A Meta-Learning Approach for Software Refactoring," *SSRN*, 2023, https://doi.org/10.2139/ssrn.4355385.

[4] S. I. Khaleel and G. K. Al-Khatouni, "A literature review for measuring maintainability of code clone," *Indones. J. Electr. Eng. Comput. Sci.*, vol. 31, no. 2, p. 1118, Aug. 2023, https://doi.org/10.11591/ijeecs.v31.i2.pp1118-1127.

[5] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, Nov. 2012, https://doi.org/10.1145/2393596.2393655.

[6] S. M. Akhtar, M. Nazir, A. Ali, A. S. Khan, M. Atif, and M. Naseer, "A Systematic Literature Review on Software-refactoring Techniques, Challenges, and Practices," *VFAST Trans. Softw. Eng.*, vol. 10, no. 4, pp. 93-103, 2022, https://doi.org/10.21015/vtse.v10i4.1275.

[7] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, Athens: IEEE, pp. 329–331, Apr. 2008, https://doi.org/10.1109/CSMR.2008.4493342.

[8] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How We Refactor and How We Document it? On the Use of Supervised Machine Learning Algorithms to Classify Refactoring Documentation," *Expert Syst. Appl.*, vol. 167, p. 114176, Apr. 2021, https://doi.org/10.1016/j.eswa.2020.114176.

[9] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, "A Survey of Deep Learning Based Software Refactoring".

[10] J. P. dos Reis, F. B. e Abreu, G. de F. Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," *Arch. Comput. Methods Eng.*, vol. 29, no. 1, pp. 47–94, Jan. 2022, https://doi.org/10.1007/s11831-021-09566-x.

[11] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pp. 91–100, 2003, https://doi.org/10.1109/CSMR.2003.1192416.

[13] M. Abebe and C.-J. Yoo, "Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review," *Int. J. Softw. Eng. Its Appl.*, 2014, http://dx.doi.org/10.14257/ijseia.2014.8.6.24'.

[14] J. Gerling, "Machine Learning for Software Refactoring: a Large-Scale Empirical Study," 2020, https://repository.tudelft.nl/record/uuid:bf649e9c-9d53-4e8c-a91b-f0a6b6aab733.

[15] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359, 2004, https://doi.org/10.1109/ICSM.2004.1357820.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010, https://doi.org/10.1109/TSE.2009.50.

[17] M. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *J. Syst. Softw.*, vol. 81, no. 4, pp. 502–516, Apr. 2008, https://doi.org/10.1016/j.jss.2007.06.003.

[18] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1106–1113. Jul. 2007, https://doi.org/10.1145/1276958.1277176.

[19] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *J. Softw. Eng. Res. Dev.*, vol. 6, no. 1, p. 3, Dec. 2018, https://doi.org/10.1186/s40411-018-0046-4.

[20] M. K. O'Keeffe and M. O. Cinneide, "Getting the most from search-based refactoring," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1114–1120, Jul. 2007, https://doi.org/10.1145/1276958.1277177.

[21] N. Nikolaidis, D. Zisis, A. Ampatzoglou, N. Mittas, and A. Chatzigeorgiou, "Using machine learning to guide the application of software refactorings: a preliminary exploration," in *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pp. 23–28, Nov. 2022, https://doi.org/10.1145/3549034.3561178.

[22] B. K. Sidhu, K. Singh, and N. Sharma, "A machine learning approach to software model refactoring," *Int. J. Comput. Appl.*, vol. 44, no. 2, pp. 166–177, Feb. 2022, https://doi.org/10.1080/1206212X.2020.1711616.

[23] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring," *arXiv*: arXiv:2001.03338. 2020, https://doi.org/10.48550/arXiv.2001.03338.

[24] M. Akour, M. Alenezi, and H. Alsghaier, "Software Refactoring Prediction Using SVM and Optimization Algorithms," *Processes*, vol. 10, no. 8, p. 1611, Aug. 2022, https://doi.org/10.3390/pr10081611.

[25] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining Software Evolution to Predict Refactoring," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 354–363. Sep. 2007, https://doi.org/10.1109/ESEM.2007.9.

[26] L. Kumar and A. Sureka, "Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 90–99, Dec. 2017, https://doi.org/10.1109/APSEC.2017.15.

[27] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*, Second edition. in The Addison-Wesley signature series. Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich: Addison-Wesley, 2019. https://archive.org/details/RefactoringImprovingTheDesignOfExistingCode1stEditionByMartinFowlerKentBeckJohnB.

[28] X. Ge, S. Sarkar, and E. Murphy-Hill, "Towards refactoring-aware code review," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 99–102, Jun. 2014, https://doi.org/10.1145/2593702.2593706.

[29] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox," *arXiv*: arXiv:2102.05201, 2021, https://doi.org/10.48550/arXiv.2102.05201.

[30] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software Engineering Challenges of Deep Learning," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 50–59, Aug. 2018, https://doi.org/10.1109/SEAA.2018.00018.

[31] M. Jeevanantham, J. Jones, "Extension of Deep Learning Based Feature Envy Detection for Misplaced Fields and Methods," *Int. J. Intell. Eng. Syst.*, vol. 15, no. 1, Feb. 2022, https://doi.org/10.22266/ijies2022.0228.51.
[32] Z. Feng *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *arXiv*: arXiv:2002.08155, 2020, https://doi.org/10.48550/arXiv.2002.08155.
[33] L. R. R. Pereira, D. L. Pereira, and R. S. Durelli, "Realizing Refactoring Prediction through Deep Learning," in *Anais do III Workshop Brasileiro de Engenharia de Software Inteligente (ISE 2023)*, BSep. 2023, pp. 7–12. https://doi.org/10.5753/ise.2023.235749.
[34] A. Almogahed, H. Mahdin, M. Omar, N. H. Zakaria, G. Muhammad, and Z. Ali, "Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems," *IEEE Access*, vol. 11, pp. 99143–99158, 2023, https://doi.org/10.1109/ACCESS.2023.3313186.
[35] A. Almogahed, M. Omar, N. H. Zakaria, G. Muhammad, and S. A. AlQahtani, "Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality," *IEEE Access*, vol. 11, pp. 28800–28819, 2023, https://doi.org/10.1109/ACCESS.2022.3218007.
[36] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, p. 421, 2008, https://doi.org/10.1145/1368088.1368146.
[37] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011, https://doi.org/10.1016/j.jss.2011.05.016.
[38] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When Does a Refactoring Induce Bugs? An Empirical Study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 104–113, Sep. 2012, https://doi.org/10.1109/SCAM.2012.20.
[39] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model Refactoring Using Interactive Genetic Algorithm," in *Search Based Software Engineering*, vol. 8084, pp. 96–110, 2013, https://doi.org/10.1007/978-3-642-39742-4_9.
[40] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, Dec. 2014, https://doi.org/10.1007/s10664-013-9256-x.
[41] R. Malhotra and A. Chug, "An empirical study to assess the effects of refactoring on software maintainability," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 110–117, Sep. 2016, https://doi.org/10.1109/ICACCI.2016.7732033.
[42] M. Mohan and D. Greer, "MultiRefactor: Automated Refactoring to Improve Software Quality," in *Product-Focused Software Process Improvement*, vol. 10611, pp. 556–572, 2017, https://doi.org/10.1007/978-3-319-69926-4_46.
[43] M. Pozza, A. Rao, A. Bujari, H. Flinck, C. E. Palazzi, and S. Tarkoma, "A refactoring approach for optimizing mobile networks," in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2017, https://doi.org/10.1109/ICC.2017.7996650.
[44] I. N. Ikhsan and M. Z. Catur Candra, "Automatically: An Automated Refactoring Method and Tool for Improving Web Accessibility," in *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, pp. 1–6, Nov. 2018, https://doi.org/10.1109/ICODSE.2018.8705894.
[45] R. T. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, "A Tool for Optimizing Java 8 Stream Software via Automated Refactoring" https://mbagherz.bitbucket.io/lab-correct-software/papers/stream-refactoring-tool.pdf.
[46] L. Cruz and R. Abreu, "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps," *arXiv*: arXiv:1803.05889, 2018, https://doi.org/10.48550/arXiv.1803.05889.
[47] R. Saheb Nasagh, M. Shahidi, and M. Ashtiani, "A fuzzy genetic automatic refactoring approach to improve software maintainability and flexibility," *Soft Comput.*, vol. 25, no. 6, pp. 4295–4325, Mar. 2021, https://doi.org/10.1007/s00500-020-05443-0.
[48] S. Tarwani and A. Chug, "Assessment of optimum refactoring sequence to improve the software quality of object-oriented software," *J. Inf. Optim. Sci.*, vol. 41, no. 6, pp. 1433–1442, Aug. 2020, https://doi.org/10.1080/02522667.2020.1809097.
[49] J. P. Sandoval Alcocer, A. Siles Antezana, G. Santos, and A. Bergel, "Improving the success rate of applying the extract method refactoring," *Sci. Comput. Program.*, vol. 195, p. 102475, Sep. 2020, https://doi.org/10.1016/j.scico.2020.102475.
[50] D. D. Pompeo and M. Tucci, "Search Budget in Multi-Objective Refactoring Optimization: a Model-Based Empirical Study," in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 406–413, Aug. 2022, https://doi.org/10.1109/SEAA56994.2022.00070.
[51] F. Durán and G. Salaün, "Optimization of BPMN Processes via Automated Refactoring," in *Service-Oriented Computing*, vol. 13740, pp. 3–18, 2022, https://doi.org/10.1007/978-3-031-20984-0_1.
[52] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci, "Many-objective optimization of non-functional attributes based on refactoring of software models," *Inf. Softw. Technol.*, vol. 157, p. 107159, May 2023, https://doi.org/10.1016/j.infsof.2023.107159.

BIOGRAPHY OF AUTHORS

Shahbaa I. Khaleel was born in Mosul, Nineveh, Iraq, She received the B.S., M.Sc. and Ph.D. degrees in computer science from Mosul University, in 1994, 2000 and 2006, respectively, assistant prof. since 2011, and finaly, prof. degree on 2021. From 2000 to 2024, she was taught computer science, software engineering and techniques in the College of Computer Sciences and Mathematics, University of Mosul. She has research in a field computer science, software engineering, intelligent technologies. She can be contacted at shahbaaibrkh@uomosul.edu.iq.
https://www.researchgate.net/profile/Shahbaa-I-Khaleel/research.

Rasha Ahmed was born in Mosul, Nineveh, Iraq, She received the B.S., degree in software engineering from the Mosul University, in 2022, and she study now Master Degree in Software Department, College of Computer Science and Mathematics, Mosul University, Iraq. She can be contacted at email: rasha.23csp16@student.uomosul.edu.iq.
https://www.researchgate.net/profile/Rasha-Ahmed-46.