Journal of Education and Science (ISSN 1812-125X), Vol: 30, No: 1, 2021 (89-102)

Refactoring for software maintenance: A Review of the literature

Rasha Gh. Alsarraj [*1], Atica M. Altaie [2]

[1, 2] Department of Software, College of Computer Science and Mathematics, University of Mosul, Iraq

E-mail: [1*] rasha.alsarraj@uomosul.edu.iq, [2] atica_altaie@uomosul.edu.iq

(Received June 26, 2020; Accepted September 05, 2020; Available online March 01, 2021)

DOI: 10.33899/edusj.2020.127426.1085, © 2020, College of Education for Pure Science, University of Mosul.

Abstract

One of the techniques to increase the value of the software quality is refactoring - the set of activities for code enhancement through altering inner structure and not altering outer behavior of code. It is a technique to clean-up the source code that decreasing the opportunities of code faults. Refactoring can be defined as one of the most significant practices for maintaining the advanced software systems. It has been indicated by the empirical studies that refactoring has positive effect on maintainability and understandability of the software systems. This study introduces a literature review of 22 researches that study and summarize the influence of refactoring and their effect on the attributes of software quality specially maintainability. Through the review, the study sums the following points: (1) applying refactoring activities will increase the values of some attributes of quality like Understandability and maintainability. (2) There are several factors that affect reconstruction activities, including cohesion, coupling, hiding of information and encapsulation, (3) Refactoring helps to improve the source code without changing the behavior of the program, (4) refactoring activates can be applied many times to the source code.

Keywords: Refactoring approaches, Refactoring tools, Refactoring challenges, Refactoring risks, Refactoring advantages, Maintainability.

إعادة البناء لصيانة البرامجيات: مراجعة أدبية

رشا غانم السراج و [2] عاتكة محمد الطائي [*1]

كلية علوم الحاسوب والرياضيات, قسم البرمجيات, جامعة الموصل, موصل, العراق

الخلاصة

اعادة البناء هي احدى تقنيات زيادة قيمة جودة البرمجيات وهي – مجموعة من الأنشطة لتحسين الشفرة المصدرية من خلال تغيير الهيكل الداخلي وعدم تغيير السلوك الخارجي. وإنها تقنية لتنظيف الشفرة المصدرية والتقليل من فرص حدوث الأخطاء داخلها. ويمكن تعريف اعادة البناء على أنها واحدة من أهم الممارسات لصيانة أنظمة البرمجيات المتقدمة. وقد أشارت الدراسات التجريبية إلى أن إعادة البناء لها تأثير إيجابي على قابلية الصيانة وقابلية الفهم لأنظمة البرمجيات. تم في هذه الدراسة تقديم مراجعة أدبية لاثنين وعشرين بحثًا من دراسة وتلخيص تأثير إعادة البناء على خصائص جودة البرمجيات ومنها قابلية الصيانة. من خلال دراسة البحوث المتعلقة باعادة البناء تبين أن (1) تطبيق أنشطة إعادة البناء سيزيد من قيم بعض خصائص الجودة مثل قابلية الفهم وقابلية الصيانة ، (2) هناك العديد من العوامل التي تؤثر على أنشطة إعادة البناء ، بما في ذلك التماسك والاقتران وإخفاء المعلومات والتغليف ، (3)

إعادة البناء تساعد على تحسين الشفرة المصدرية دون تغيير سلوك البرنامج (4) يمكن تطبيق أنشطة إعادة البناء عدة مرات على الشفرة المصدرية.

الكلمات المفتاحية: نماذج اعادة البناء, ادوات اعادة البناء, تحديات اعادة البناء, مخاطر اعادة البناء, فوائد اعادة البناء, قابلية الصيانة.

Introduction

Smart development and regular maintenance are required to achieve good quality. A lot of activities related to quality assurance might be used such as refactoring, code walks, review, and testing. They might be utilized in the case when the software is produced for the first time or in the case when previously present resources were utilized such as design templates or source codes. Based on Fowler [1], the refactoring can be defined as the process used to alter software's internal layout without making alterations in external behavior. Also, the process might be utilized via software engineers by means of patterns of code time and again [2].

Throughout years, empirical studies indicate positive relations between code quality metrics and refactoring operations (e.g., [3], [4],[5]), such evidences indicating that the refactoring might be specified as first-class concern with regard to software developers. Yet, to decide what and when (in addition to knowing why) to refactor, was a challenge to developers. The teams of software development must not be simply refactoring their software systems on demand, or deciding not to refactor a code which results in technical debt, since any activity of refactoring comes with certain cost [6]. In addition to all efforts made by the community to provide the tools which will refactor the source code automatically, identifying the refactoring opportunities (for instance, identify the methods or classes which must be refactored) is a stage of high importance that comes before the process of refactoring. The aim to identify related to refactoring opportunities, which is presently based on intuition and expertise of developer, must be supported through advanced recommendation algorithms. Modeling the entire context which is faced via the developed in the case when deciding what to refactor, has been difficult problem. Studies were experimenting with various methods to recommend refactoring, for instance, code smells detection strategies [2], invariant mining, logic meta programming, search-based, and pattern mining.

Section two of this study provides the definitions of refactoring. Section three provides the refactoring challenges. Section four will provide the refactoring decision. Section five is providing explanation related to the risks and benefits of refactoring. Section six will provide the literature review, while section seven will provide the main conclusions.

2. Refactoring Definition in Practice

Refactoring can be defined as one of the software maintenance activities to improve the code internal structures, whereas maintaining its external behavior [7]. In the past 10 years, a lot of studies were indicating that the refactoring might be reducing the software complexity, improving the developer comprehensibility as well as improving the start-up time and memory efficiency [8]. Therefore, the developers were encouraged for performing the operations of refactoring regularly [1]. Based on a study by Mens et al. [9], one might divide the process of refactoring as follows [8]:

1. Identifying the code entities which require refactoring, the term code entities indicates the classes on the object-oriented systems. Also, the major utilized method for detecting code entities which require refactoring is anti-patterns' detection and/or code smells [9]. In addition, the anti-patterns are bad design choices for recurring the design problems. Generally, they are provided through inexpert developers, also representing major pitfalls in the software development. In a previous study, Coplien

and Harrison [10] provide something with might be excellent approach, yet might be badly backfiring when utilized. The difference between code smells and anti-patterns is that the first ones are local problems indicating the existence of general design problems (i.e., anti-patterns). For instance, the low cohesions, large class, and long methods, are symptoms regarding Blob Class anti-pattern [1]. It must be indicated that using anti-patterns for identifying the code entities which must be refactored vary based on the applications' domain. For instance, in the case of focusing on mobile and/or embedded systems, one may consider that the anti-patterns are associated to energy efficiency along with conventional design anti-patterns, for providing complete design solution.

2. Determining the major adequate refactoring that must be utilized. A study by Fowler suggests a catalog of 22 refactoring as well as informally associated the anti-patterns to refactoring [1] The informal guidelines were used in the past studies for generating refactoring opportunities for removing the anti-patterns [11] in a semi-automated fashion.

3. Ensuring that the utilized refactoring is preserving behavior. Generally, current approaches of refactoring adopt the refactoring post- and pre- conditions suggested via Opdyke [8] for preserving the refactored system's behavior.

4. Measuring the impact of refactoring used on the required quality attributes (for instance, complexity, flexibility, and understandability.). "You can't control what you can't measure". As soon as applying set of refactoring, one must have the ability for assessing its effect on the design quality. Mainly, studies are using code and object-oriented metric suites, in addition to the anti-pattern's/design patterns occurrences for evaluating the design improvements.

5. Maintaining the consistency between the software design as well as the other software artifacts (for instance, tests, documentation, and so on.). Following using and estimating set of refactoring, the risks of impacting the consistency between the source codes as well as the other software artifacts involving the documentation, design models, and test suite is high. Therefore, it has been suggested for counting the approaches for maintaining the consistency [8].

3. Refactoring Challenges

There are various challenges related to adopting refactoring [12,13], 28% of the developers indicated certain inherent challenges including the work on large code bases, large amounts of the inter-component dependencies, the requirements of coordination with the other teams and developers, also the difficulties to ensure program's correctness following refactoring. 29% of the developers indicate the absence of tool support for refactoring change integration, code review tool targeting refactoring edits, as well as advanced refactoring engines where the user may simply define the new refactoring types. In addition, the difficulties to merge and integrate following refactoring sometimes encourage the individuals from doing refactoring [14]. The version control systems which they utilize have been sensitive to move and rename refactoring, also making it difficult for developers to understand the code change history following refactoring.

The next quotes describe the difficulties of refactoring change integrations in addition to code reviews following refactoring: "Cross-branch integration has been the major issue [15]. Also, developers have such issue each time they are fixing the bugs or when refactoring anything, even though that in such case it has been especially difficult, since the refactoring moved files, that prevents cross-branch integration patches from being used." "Typically, refactoring increases the number of lines/files included in the check-in, which burdens the reviewers of code. It also increases the odds in which the changes are going to collide with the changes of someone else." A lot of participants indicate that in

the case when the regression test suite has been insufficient, there will be no safety nests to check refactoring's correctness. Therefore, this prevents developers from initiating refactoring efforts [16].

If there are extensive unit tests, the (it's) great, (one) must be refactoring unit tests as well as run them, and doing certain sanity testing on the scenarios. In the case when there are no tests, then (one) must go from the known scenarios and ensuring all works. Furthermore, in the case when there are no sufficient documentations for the scenarios, refactoring must be achieved. The inherent in addition to technical challenges regarding refactoring indicated via participants, maintain backward compatibility sometimes discouraging them from starting refactoring efforts [13].

4. Refactoring Decision

The developers perform refactoring based on the signs of code, helping to make a decision on the refactoring (Fig. 1) [12,16]. 22% have stated insufficient readability; 11% have stated that it is difficult to repurpose the available code for various cases and predicted features; 11% have stated insufficient maintainability; 13% have stated a duplication in the code; 9% have stated that it is difficult to test the code with no refactoring; 8% have stated that the performance is slow; 5% have stated that there are dependencies to modules of other teams; and 9% have stated code of old legacy which they need to operate on. 46% of the developers have stated that they perform the refactoring in the contexts of feature additions and bug fixing, and 57% of them have indicated that the refactoring is regulated with the immediate visible, concrete needs of the variations which they have to implement in a limited period of time, instead of the possibly indefinite advantages of the long-term maintainability. Additionally, over 95% of the developers perform the refactoring over all of the milestones and not only in the MQ milestones—a period which has been specified for fixing the bugs and cleaning up the code with no responsibility for the addition of new features which means the refactoring effort pervasiveness [16]. Base on the self-reported data, the developers spend approximately 13h. each month to work on the refactoring, and that is approximately 10% of their works, in the assumption that the developers are working approximately 160 h. monthly [13,16].

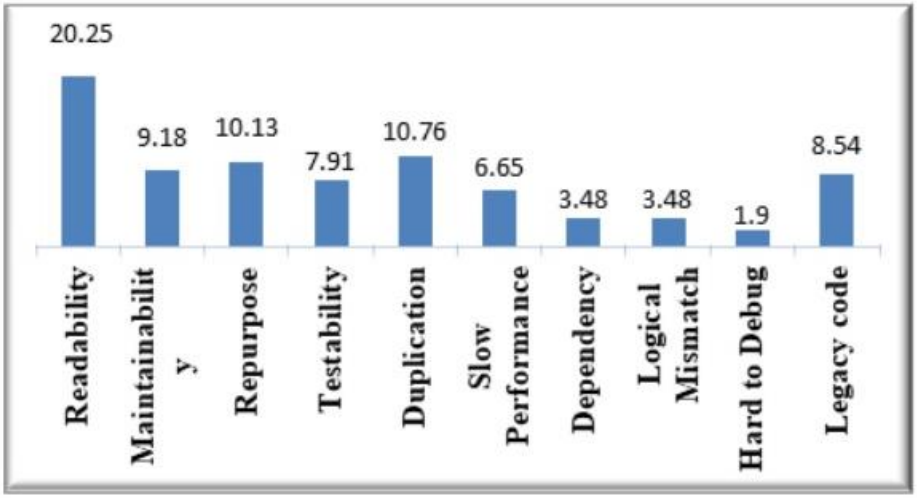

Fig. 1. The symptoms of code that help developers initiate refactoring

5. Refactoring Risks and Advantages

Based on the developers' experience, the related risks of the refactoring are, code churns, regression bugs, time which is taken from the other tasks, merge conflicts, difficulty to do code reviews following the refactoring, and over engineering risks [12]. Fig. 2 includes the rate of the developers that have stated every one of the specific risk factors. It should be noted that total summation is up to 100% as one of the developers would be mentioning several risk factors. 76% of participants have considered

that the refactoring is combined with risks of the introduction of functionality regressions and subtle bugs; 11% state the fact that the code merging is difficult following the refactoring; while 24% have mentioned the increase in the costs of testing. "The main risk is the regression, usually due to the misunderstanding of the cases of the subtle corners in original code and not accounting for those corners in refactored codes." "The Over-engineering—developer can produce unnecessary architectures which aren't required by any one of the features; however, each chunk of code must adapt to it." "It is difficult to measure the refactoring value. How It is possible to measure a bug value which has never existed, or time which is saved on a later unspecified feature? How can such value bubble up to the management? Due to the fact that there are no ways for placing the exact value on refactoring practice, it is hard to justify to the management [16].

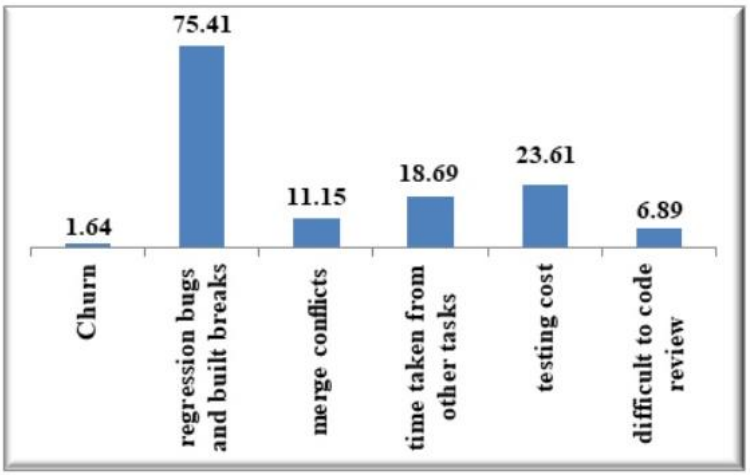

Fig. 2. The risk factors associated with refactoring

The benefits observed by the developers from the refactoring are the enhanced readability, enhanced maintainability, enhanced performance, fewer bugs, duplicate code reduction, code size reduction, enhanced extensibility & higher simplicity for adding new features, enhanced testability, decreased time to market, enhanced modularity, and so on, as shown in Fig. 3 [16,17].

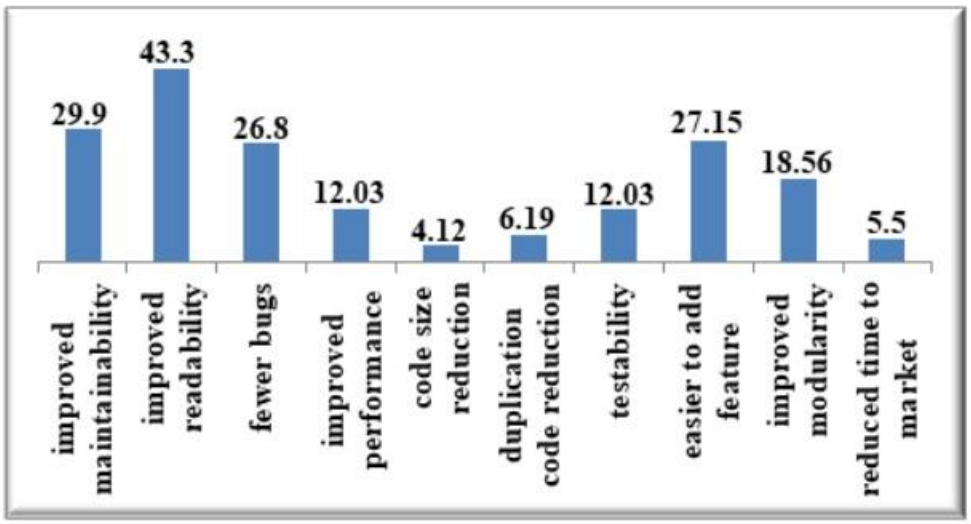

Fig. 3. Various types of refactoring benefits that developers experienced

6. Literature Review of Refactoring

The life-cycle of software is highly dependent on maintenance. Throughout the maintenance, the code might be altered many times, that might be deteriorating the quality of the code. Therefore, the software's maintainability is a main focus for software industries, one of the major approaches is refactoring [18].In the present survey, the objective is the aggregation of the patterns and the information on the researches which are carried out on the refactoring for the maintenance of software and trends which are not discovered earlier.

In 2010, Hegedus et al examine how quality attributes including maintainability, testability and error proneness can be predicted depending on metrics that can be measurable in the source code, like cohesion , line of codes, and complexity and inspect their effect of each refactoring approach on the computed metrics. Furthermore, Applying refactoring of code is definitely a reasonable and efficient activity in facilitating maintenance, and it increases the value of quality attributes for the software, if refactoring activates are used properly at a suitable time [19]. In the same year, Mubarak studies the effect of coupling between classes on refactoring and maintainability and how the project manager can estimate maintenance effort and/or refactoring effort needed to work the project correctly. The results recommend that the maintenance changes can applying new method called "peak and trough‟ to calculated effect and related of experimental data of refactoring for the software. The results also recommend to applying the refactoring activities after a regular change of maintenance activities and the advantages of removing coupling between classes on decreasing the refactoring effort precisely with refactoring of package and giving warning data for potential of refactoring [20].

In 2011, Singh and Kahlon introduce a measure to classify faulty classes which are cohesion, hiding of information and encapsulation. Then, the researchers propose a statistical method to examine the correlation between these measures and source

code smells. Firefox is considered as a case study to prove the approach and the results display high precision in classifying the classes that are faulty and applying refactoring activates on the faulty classes. These results of refactoring will assist the developer in testing and maintenance phases of software life cycle [21]. Present's sequential processes are future's parallel processes. So in the same year, Dig introduces a new method to applying refactoring techniques for analyzing and transforming current processes. The method suggests to alter multiple lines of source code and eliminate error-prone because programmers require ensuring parallel processes to increase software performance, maintainability and portability. The research also presents a set of tools which supports many activities of refactoring techniques including increasing scalability and maintainability of parallel processes, making processes thread-harmless and increasing throughput of sequential processes [22].

Sometimes the programmer needs to apply the refactoring multiple times to the source code. So in 2012, Meananeatra proposes a method to recognize a best sequence of refactoring that meets many measures which are maintainability factor, the total number of faults eliminated, the dimension sequence of refactoring, and the total number of changed program components. Furthermore, the authors estimates that the results tend to decrease cost and time of maintainability, and improve the quality of software. The method of research does not produce all refactoring sequence at one time but regularly detect a graph for sequences and use refinement method to remove opposite sequence of refactoring in order to find a best sequence to be implemented [23]. In the same year, Villavicencio classifies the technique of refactoring into : understanding and efficiency, the first is useful in maintenance and the second is for implementation. Understanding and efficiency , the former is considered the opposite of the later. Therefore, refactoring techniques in forward and reverse can be noticed as forms summarizing the information on how to arrange the source code for increase the efficiency and easy maintenance [24].

In 2013, Fujiwara et al propose a methodology to evaluate refactoring processes from archives of version which is improves the quality including ease of maintainability. The methodology is semi-automatically implemented by inspecting in archives of software based on two algorithm which are UMLDiff "UML difference" and SZZ "Sliwerski, Zimmermann, and Zeller". The author adopts Columba project as a case study and the result displays that the occurrence of defect is reduced after many circles of refactoring and increases the factor of maintainability through three variables : frequent of refactoring, frequent of fix and defect density [25]. code clone is a duplicated code and may be a malicious code that require to be eliminated to improve maintainability. So in the same year, Zibran and Roy introduced a model for predict the effort need to eliminate code clone through refactoring and proposed an approach called CP "constraint programming" for best refactoring scheduling of code clone elimination. Case studies are taken to examine the effort model and scheduler of clone code refactoring, results compared with other scheduling approaches and CP-depended model shown outperforms other models [26].

In 2014, Chaparro et al introduce a technique called Refactoring Impact Prediction (RIP) to study the effect of refactoring processes on the quality metrics of source code. Using this technique , developers can evaluate refactoring chances in the maintenance tasks of software, also it permits developers to compare the value of metric deviations caused by the processes of refactoring, particularly when refactoring contains many transformations and evaluates conflicting metrics of the source code [27]. In the same year, Steidl and Eder propose to eliminate maintainability defects, reuse of optimal code and long functions, that require less effort to refactor. The research provides the developers a point to enhance the quality of the software. With a case study for java software, the research calculates and assesses the advantage of recommendation depending on feedback from developers and specifies effect of external factors on the process of maintainability defect elimination in software development [28].

In 2015, Ah-Rim et al introduce term of MIS "maximal independent set" allows the developer to recognize many processes of refactoring which can implemented at the same time and each MIS has a collection of refactoring paths that calculates delta table which is the values of maintainability for each primary path. In each circle of the process of refactoring, many operations can be applied to increase maintainability value through sets of MISs. The proposed model is implemented on many of case studies and the results display that the model can increase maintainability factor. Furthermore, the developer can apply many circles of refactoring at the same time [29]. In the same year, Kannangara and Wijayanayake study the effect of refactoring approaches based on multiple measures to increase the quality of software. Ten approaches have been selected and assessed quality through five external factors and five internal factors including index of maintainability. The results didn't show any enhancement on software quality after applying refactoring processes for external factors and show good enhancement of maintainability index in the quality of source code that refactored for internal factors and no enhancement for other internal factors [4].

In 2016, Mohan and et al, compare 4 distinct methods of refactoring with the use of automated tool of software refactoring. The weighted summation values of the metrics have been utilized for forming a variety of the fitness functions driving the process of the search in the direction of specific software quality aspects. The measures are integrated for measuring the abstraction, inheritance, and coupling and a 4^{th} function of fitness has been suggested for measuring the technical debt reduction. Those four functions have been compared to one another with the use of three distinct searches on six distinct programs of open source; four out of the six of the programs have shown a higher enhancement in the function of the technical debt following the process of the search based refactoring. The results have shown that this function has been beneficial to assess quality enhancement [30]. In the same year, Malhotra and Chug examine the refactoring impacts on the maintainability with the use of 5 proprietary software systems. The internal attributes of the quality have been evaluated with the use of

the design measures suite while the external attributes of the quality like the levels of the understandability, abstraction, extensibility, modifiability, and reusability have been evaluated by expert opinions. The original software versions have been compared to the versions that have been refactored and quality attribute changes have been mapped to the maintainability. The results have revealed that there has been a significant improvement of software quality refactoring and enhancement in the life of the software. It has been discovered as well, that although the refactoring is highly tedious and can be introducing errors in the case where they are not implemented with the maximum care, it remains advisable for the frequent refactoring of code for increasing the maintainability. The results of this study have been beneficial for the projection of the management to identify the refactoring opportunities at the same time as keeping an ideal balance between the reengineering and the over-engineering[31].

In 2017, Mohan and Greer proposed a novel method for the automated software maintenance. This tool is capable of performing 26 distinct refactoring processes. in addition to that, it contains a wide range of selections of the measures for evaluating the refactoring impact on software and 6 distinct search based methods of the optimization for enhancing software. Such tool includes mono-objective as well as multi-objective methods of searching for the enhancement of the software and it has been entirely automated. The researchers have explained the variety of the tool's abilities, as well as its unique aspects, in addition to presenting a number of the study results from the experiment. The distinct metrics have been tested over 5 distinct code-bases for the deduction of the most efficient measures for the general enhancement of the quality. It is found that metrics relating to higher specificity aspects of code are of a higher usefulness to drive the changes of searching. Mono-objective genetic algorithm has been tested as well against multi-objective approach, for observing the degree of the comparability of results that have been obtained with 3 distinct aims. In the case of the comparison of the optimal solutions of every one of the separate objectives the multi-objective method produces proper quality enhancement in a shorter period, which allows a fast cycle of maintenance [5]. In the same year, Alvarado enhances the automated refactoring by taking into consideration new dimensions: (a) developer's task contexts for prioritizing the related class refactoring; (b) test efforts for improving the cost of the testing following the refactoring; (c) conflict awareness of the refactoring for the reduction of the efforts of refactoring; (d) energy preservation for improving the consumption of the energy of the mobile applications following the process of the refactoring. The author has suggested 4 methods, namely: (i) ReCon, leveraging task context of the developer for the prioritization of the refactoring of the classes which are associated with the activities of the developer. By the use of the ReCon, the developer gains the ability of removing a median of 50% anti-patterns throughout the normal tasks of coding, with no disruption to their work-flow. (ii) TARF controls for testing efforts while refactoring. The Results have shown that the TARF is capable of reducing a median of 48% of testing efforts of a system following the refactoring. (iii) RePOR, to efficiently refactor the scheduling, resulting in an 80% reducing of the efforts of refactoring and time of execution. (iv) EARMO, which is an automated method to refactor the mobile applications, capable of the removal of 84% of the anti-patterns and extending the battery life for the devices by about 29 min. (for a multi-media application which runs continuously uncommon scenario). He applies and validates his suggested methods on a number of the open-source systems for demonstrating their effect upon the quality of the design with the use of the common models of the quality, and the feedback from a number of the authors of the researched systems [8].

In 2018, Kula et al conduct an experiential research for exploring the correlation between the Application Programming Interfaces (API) refactoring and breakage processes which have been based upon the actual utilization of the API by the clients. They are capable of distinguishing between the APIs of the library based upon their client-utilization (known as the client-used API) for the purpose of getting a deeper knowledge about the level to which the breakages of the API may be associated

with the refactoring actions. This study covers more than 9700 of the breaking classes and approximately 12900 of the refactoring processes from 8 common libraries of Java, with every one of the libraries which have about 10∼38 of the successive releases. They have noticed the following points: (a) the maintainers of the library have lower likelihood of breaking the client-used APIs in comparison with other library classes, (b) the found refactoring processed breaks only < 37% of the client-utilized APIs, (c) the remaining (63%) breakages of the API are triggered with the issues of the maintenance which will be possibly involving more complicated refactoring processes. (d) Simple refactoring processes (such as rename approach, move approach, and move field) have been applied less often to the client-used classes of the APIs, in comparison with other class types [32]. In the same year, Mohan explores a Search-based software maintenance (SBSM), develops and proposes a new tool for fully automated Java software maintenance with the use of the multi-objective, mono-objective, and many-objective methods of the searching. This tool has been supplied by several refactoring processes and metrics and is entirely configurable. It can be found available on-line to be used and may be utilized for the researching purposes or as one of the maintenance tools for assisting with improving and maintaining Java software. The tool measures the priorities of classes which have been refactored in refactored solution. It also measures the code coverage of the refactoring solutions generated in refactoring tool. The author has presented a systematic current opportunities' analysis with the SBSM. The variety of the tools which are available presently have been analyzed and examined. The variety of the search-based optimization methods have been examined as well and the variety of the searches have been compared with one another for the analysis of the benefits and drawbacks of the variety of methods. The disadvantages of the available methods have been analyzed and either addressed or outlined [7].

In 2019, Fengrong and et al provide the mechanisms related to clone code as well as refactoring. Firstly, the clone code can be defined as a fragment of code which is similar or identical in semantics or syntax, that is affecting the maintenance and development of software. After that, the major approaches for present clone code re-factoring will be put to comparison and analysation. In addition, the majorly applied methods for clone refactoring might be categorized into the next categories: evolutionary analysis-based method, program dependency graph-based method, abstract syntax tree-based method, metric-based method, refactor-ability related to clone code evaluation, approach on the basis of extracting the clone metric for re-factoring, an approach on the basis of revenue-cost assessments, and an approach on the basis of evolutionary coupling and measuring. Therefore, the associated tools regarding clone code re-factoring might be elaborated, also their benefits and drawbacks will be indicated. There are majorly 4 tools for refactoring in the current time, Ref-Finder, FaultBuster, JDeodorant, and SPCPMine. Lastly, the drawbacks of clone code re-factoring will be indicated [18]. In the same year, Mohan and et al , describe investigating a many-objective genetic algorithm which has been utilized for the automation of the process of the software refactoring, which have been implemented as a Java tool, Multi-Refactor. The method and the tool have been assessed with the use of a group of the open source programs of Java. This tool contains 4 distinct Software looking measures at the quality of the software in addition to the measurements of the priority of the code, element recentness, and refactoring coverage. The algorithm of the many-objective is involved with the combination of the 4 aims for improving software in a holistic way. An experimentation has been created for the comparison of many-objective method with the mono-objective method which utilizes only one objective for the measurement of the quality of the software. A variety of the objective permutations have been tested as well, and compared for the purpose of seeing how efficiently the variety of the aims may operate combined in a multi-objective method of refactorings. The 8 methods have been experimented on 6 distinct open source programs of Java and results are as follows: The method of the many-objective has been discovered to provide more sufficient objective score values on average compared to mono-objective method and was faster. None-the-less, the element recentness and priority aims have been discovered to have lower rate of success in many-objective/multi-objective

set-ups in the case where they have been utilized in combination. The researchers have reached a conclusion that a many-objective method is proper and sufficient to optimize the automated refactoring for the improvement of the quality. This means that other aims doesn't excessively reduce the enhancements' quality, however, it is less efficient for these objectives compared to the case where they are utilized in the mono-objective method [33].

In 2020, Morales and et al have carried out an empirical research for investigating it the structure of the automated refactoring code has an impact on the system understandability throughout the tasks of the comprehension. (a) they have conducted a survey of 80 developers, as they asked them to recognized from a group of 20 changes of refactoring in the case where they have been produced by a tool or by the developers, and also to provide the rating of the changes of refactoring based on their quality of design; (b) they have requested 30 developers to carry out the tasks of the code comprehension on 10 systems which have been refactored via a freelancer or through automated tools of refactoring. They have performed a measuring of performance of the developers with the use of NASA task load index for their efforts, the time which they have spent carrying tasks, and their percentage values of the accurate answers. In spite of the current limitations if the technology, their findings have shown that it's reasonable expecting the refactoring tool to be matching the code of the developer. In fact, the results have shown that for 3/5 of the studied types of the anti-patterns, the developers have no ability for recognizing the refactoring origin (in other words, whether or not it has been carried out via an automatic tool or a human). In addition to that, they have noticed that the developers don't have a preference for the human refactoring processes over the automated processes of refactoring, except the case of refactoring classes of the Blob; and the fact that that there has not been any statistically significant distinction between the impacts on code understandability of the human and automated refactoring processes. They conclude that the automated processes of the refactoring may be equally efficient to the manual refactoring processes. None-the-less, for the complicated types of the anti-pattern, such as Blob, the perceived quality which has been accomplished by the developers is a little better [34]. So in 2020, Aniche and et al investigate machine learning (ML) approaches' effectiveness in the prediction of the refactoring processes of the software. Particularly, they have trained 6 distinct ML methods (such as the Naïve Bayesian, Logistic Regression, Decision Trees, Support Vector Machine (SVMs), Neural Networks (NNs) and Random Forest) with a data-set which comprises more than 2 million refactoring processes from 11149 of the real-world projects from F-Droid, Git-Hub, and Apache, eco-systems. The resultant models have predicted 20 distinct refactoring processes at method, class, and variable-levels with a precision which is usually over 90%. The outcomes of this research have shown that (a) Random Forest is the optimal model for the prediction of the software refactoring, (b) the metrics of the ownership and process appear to be playing a vital part in creating more efficient models, (c) the models are well generalized in a variety of the contexts [35]. Table 1 explains the summary of aforementioned.

Table 1: Summary of the studies

Year	authors	Factor study	Quality attributes effected
2010	Hegedus et al [19]	cohesion , line of codes, and complexity	Refactoring, maintainability, testability and error proneness
	Mubarak [20]	Coupling	Refactoring and maintainability
2011	Singh and Kahlon [21]	Encapsulation, information hiding, cohesion, and estimating faulty classes	Refactoring and maintainability
	Dig [22]	A multiple circles of refactoring	Maintainability portability, productivity, scalability and performance.
2012	Meananeatra [23]	A multiple circles of refactoring	Maintainability, the total number of faults eliminated, the dimension sequence of refactoring, and the total

			number of changed program components
	Villavicencio [24]	selecting a best sequence of refactoring	Maintainability, efficiency, understandability and removing bad smells
2013	Fujiwara et al [25]	Version archives	Refactoring and maintainability
	Zibran and Roy [26]	Eliminating code clone	Refactoring and maintainability efforts
2014	Chaparro et al[27]	RIPE "Refactoring Impact PrEdiction"	Code quality metrics ((RFC, CBO, DAC, MPC, LOC, NOM, CYCLO, LCOM2, LCOM5, NOC, DIT)
	Steidl and Eder [28]	Eliminating code defects, reuse of optimal code and long functions	Refactoring and maintainability
2015	Ah-Rim et al [29]	MIS "Maximal Independent Set" applying multiple processes of refactoring simultaneously.	Refactoring and maintainability
	Kannangara and Wijayanayake [4]	Assessing the effect of refactoring approaches based on multiple measures	Refactoring and Maintainability Index
2016	Mohan et al [30]	Measure of coupling, abstraction, inheritance	Refactoring and maintainability.
	Malhotra and Chug [31]	Abstraction level, understandability, extensibility, modifiability, and re-usability	Refactoring and maintainability
2017	Mohan and Greer [5]	Assessing the impacts of the approaches of refactoring based upon multiple metrics like QMOOD, CK and optimizing based on six different search algorithms.	MultiRefactoring and maintainability
	Alvarado [8]	prioritizing the refactoring of classes, Refactoring method which has been based upon task Context (ReCon), Refactoring method which has been based on Partial Order Reduction (RePOR), Testing-Aware ReFactoring method (TARF), Energy-Aware Refactoring method for the MObile applications (EARMO)	Refactoring, Refactoring efforts and maintainability
2018	Kula and et al [32]	Exploring the correlation between the Application Programming Interfaces refactoring processes and breakages based upon the actual utilization of the API by the clients	Refactoring and Maintainability
	Mohan [7]	Tool equipped with numerous refactoring activities and metrics based on Search-based software maintenance (SBSM)	Refactoring and Maintainability
2019	Fengrong et al [18]	Clone code refactoring	Refactoring and maintainability
	Mohan and Greer[33]	Measure of Code priority, refactoring coverage and element recentness ,optimizing automated refactoring.	MultiRefactoring and maintainability
2020	Morales et al [34]	"Mimicking humans on refactoring tasks" (RePOR), an automated refactoring approach based on partial order reduction techniques.	Refactoring, understandability and maintainability

	Aniche [35]	**Six different machine learning algorithms which are (Logistic Regression, Naive Bayes, Support Vector Machine, Decision Trees, Random Forest, and Neural Network)**	**Refactoring, understandability and maintainability**

7. Conclusion

Refactoring is an activities which increases the quality of software and permits software engineers to repair code which is difficult to be maintained. There are many approaches and tools to apply refactoring activities. The approach and the selected tool are based on the nature of software. This study introduces a literature review of relevant researches identifying code refactoring activities and their effect on software maintainability published from the year of 2010 to 2020. The paper discusses and summarizes many researches in the domain of code refactoring depending on a set of criteria. Among them are line of codes, cohesion , coupling, complexity, encapsulation, information hiding, estimating faulty classes, a multiple circles of refactoring and so on. It also investigates their effect on software quality attributes and states the important factors which must be occupied when building a tool for refactoring.

The paper concludes with the following: (1) Applying refactoring activities will increase the values of some attributes of quality like Understandability, maintainability and testability (2) There are several factors that affect reconstruction activities. (3) Refactoring helps to optimize code without changing software behavior. (4) Refactoring activates can be applied many times to the source code.

This survey is useful as an introduction to researchers who aim to work in the area of software refactoring with regard to software maintenance and will allow them to gain an understanding of the present landscape of research and the insights collected. This study is hoped to help and inspire suggesting future improvements in the area of refactoring approaches.

Acknowledgments

The authors would like to express their thanks to University of Mosul/ College of Computer Sciences and Mathematics for the facilities provided by them, which were very helpful in improving this work's quality.

References

1. Fowler M., Beck M., Brant K., Opdyke j., Roberts W., "Refactoring-improving the design of existing code". 1st ed. Addison-Wesley; 1999.
2. Singh, S. and Kaur S., "A systematic literature review: Refactoring for disclosing code smells in object oriented software." Ain Shams Engineering Journal 9 (2017): 2129-2151.
3. AlOmar E. A., Mkaouer M. W., Ouni A. and Kessentini M., "Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities," in Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2019), 2019, pp. 300–311.
4. Kannangara S. H. and Wijayanayake W., "An Empirical Evaluation of Impact of Refactoring on Internal and External Measures of Code Quality", International Journal of Software Engineering Applications, (IJSEA), Vol. 6, No. 1, 2015.

5. Mohan M., Greer D.," MultiRefactor: Automated Refactoring to Improve Software Quality", International Conference on Product-Focused Software Process Improvement, pp 556-572 , 2017, Springer
6. Kruchten P., Nord R. L. and Ozkaya I., "Technical debt: From metaphor to theory and practice," IEEE Software, vol. 29, no. 6, pp. 18–21, 2012.
7. Mohan M., "Automated Software Maintenance Using Search-Based Refactoring ", PHD. Thesis, Queen's University Belfast, 2018.
8. Alvarado R., "AUTOMATED IMPROVEMENT OF SOFTWARE DESIGN BY SEARCH-BASED REFACTORING", PhD thesis, UNIVERSITÉ DE MONTRÉAL, 2017.
9. Mens T. and Tourwé T., "A survey of software refactoring," Software Engineering, IEEE Transactions on, vol. 30, no. 2, pp. 126–139, 2004.
10. Coplien J. O. and Harrison N. B., "Organizational Patterns of Agile Software Development", 1st ed. Prentice-Hall, Upper Saddle River, NJ (2005), 2005.
11. Ouni A., Kessentini M., Sahraoui H., and Boukadoum M., "Maintainability defects detection and correction : a multi-objective approach," Automated Software Engineering, vol. 20, no. 1, pp. 47–79, 2013.
12. Sharma T., Suryanarayana G., and Samarthyam G., " Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective", IEEE Software, 2015.
13. Khanam Z., "Barriers to Refactoring: Issues and Solutions", International Journal on Future Revolution in Computer Science & Communication Engineering, Volume: 4 Issue: 2, 2018.
14. Brun Y., Holmes R., Ernst M. D., and Notkin D., "Proactive detection of collaboration conflicts," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ser. ESEC/FSE '11. New York, NY, USA: ACM, pp. 168–178, 2011.
15. Bird C. and Zimmermann T., "Assessing the value of branches with what-if analysis," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ser. FSE '12. New York, NY, USA: ACM, pp. 45:1–45:11, 2012.
16. Kim M., Zimmermann T. and Nagappan N., "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," IEEE Trans. Software Eng., vol. 40, no. 7, pp. 633–649, 2014.
17. Ouni A., Kessentini M., Sahraoui H., Inoue K., Deb K., "Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study", ACM Transactions on Software Engineering and Methodology, Volume 25, Issue 3, 2016.
18. Fengrong Z., Liping Z. and Junqi Z., "Research on the Tools of Clone Code Refactoring", ICMSS: Proceedings of the 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, Pages 27–31, 2019 .
19. Hegedus G., Hrabovszki G., Hegedus D. and Siket I., "Effect of object oriented refactorings on testability, error proneness and other maintainability attributes", In Proceedings of the 1st Workshop on Testing Object-Oriented Systems, ACM, 2010.
20. Mubarak A., "An Empirical Study of Package Coupling in Java Open-Source", PhD thesis, Brunel University, School of Information Systems, Computing and Mathematics, 2010.
21. Singh S. and Kahlon K.S.," Effectiveness of Encapsulation and Object-oriented Metrics to Refactor Code and Identify Error Prone Classes using Bad Smells", ACM SIGSOFT Software Engineering Notes, Vol. 36, No. 5,September 2011.
22. Dig D., "A refactoring approach to parallelism", IEEE Software, Vol.28, No.1, 2011.
23. Meananeatra P., "Identifying refactoring sequences for improving software maintainability" In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, 2012.
24. Villavicencio G., "A new software maintenance scenario based on refactoring techniques" In 16th European Conference on Software Maintenance and Reengineering (CSMR), Hungary, IEEE, 2012.

25. Fujiwara K., Fushida K., Yoshida N. and Iida H., "Assessing refactoring instances and the maintainability benefits of them from version archives", pages 313–323. Springer, 2013.
26. Zibran M. F. and Roy C. K., "Conflict-aware optimal scheduling of prioritized code clone refactoring", IET Software, 2013.
27. Chaparro O., Bavota G., Marcus A. and Di Penta M., "On the impact of refactoring operations on code quality metrics," in Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), 2014.
28. Steidl D. and Eder S., "Prioritizing Maintainability Defects by Refactoring Recommendations," in Int'l Conf. on Program Comprehension, 2014
29. Ah-Rim H., Doo-Hwan B., Sungdeok C.,"An efficient approach to identify multiple and independent Move Method refactoring candidates", Information and Software Technology, vol. 59, pp. 53-66, 2015.
30. Mohan M., Greer D. and McMullan P., "Technical debt reduction using search based automated refactoring". Journal of Systems and Software, 120, 183–194, (2016).
31. Malhotra R., Chug A., "An Empirical Study to Assess the Effects of Refactoring on Software Maintainability", Intl. Conference on Advances in Computing, Communications and Informatics (ICACCI), 2016.
32. Kula R. G., Ouni A., German D. M. and Inoue K., "An empirical study on the impact of refactoring activities on evolving client-used APIs", Information and Software Technology, 93, 186–199, 2018.
33. Mohan M. and Greer D., " Using a Many-Objective Approach to Investigate Automated Refactoring", Information and Software Technology, Technology, Vol 112, pp 83-101, 2019.
34. Morales R., Khomh F. and Antoniol G.," RePOR: Mimicking Humans on Refactoring Tasks. Are We There Yet?", Journal of Empirical Software Engineering (EMSE), 2020, Springer.
35. Aniche M., Maziero E., Durelli R. and Durelli V., "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring", computer science, ArXiv, 2020.