Journal of Education and Science (ISSN 1812-125X), Vol: 32, No: 01, 2023 (71-80)

# Software Code Refactoring: A Comprehensive Review

**Hiba Muneer Yahya[1*], Dujan B. Taha[2]**

[1*,2] Softwares Department, College of Computer Sciences & Mathematics, University of Mosul, Mosul, Iraq

E-mail : [1*]hibamoneer@uomosul.edu.iq, [2]dujan_taha@uomosul.edu.iq

(Received November 29, 2022; Accepted February 15, 2022; Available online March 01, 2023)

DOI: 10.33899/edusj.2023.137163.1298, © 2023, College of Education for Pure Science, University of Mosul.


---

**Abstract**:
The complexity of software has increased because of the development as well as the difficulty of requirements during the development of software, or to add new features that eventually lead to reduce the quality of the software as a whole. Software refactoring can be defined as included processes in the maintenance period of a software life cycle, and it is a technique to clean the software code from code bad smell and to improve the internal structure of the software, in addition to increasing the quality of software by using a set of activities without changing the external behavior of a software. Researchers have been developing techniques to reform software during the code or design standard to decrease the effort and time required for maintenance processes. This paper provides a systematic review of the literature for 17 studies of code standards, An automatic search is utilized in the digital libraries to look for the relevant studies that were published from 2014 to 2021, the best five researchers are chosen in this subject, five studies or less are chosen for each depending on the number of reference in the database of scientific sites, or using an approach or a new method to get good results. Eventually, each paragraph is analyzed and mentions the method or algorithm used in rebuilding software, further the aims, and the result for each paper.

**Keywords:** Refactoring, Software Development, Code Smell

## إعادة بناء كود البرامج: مراجعة شاملة

**هبه منير يحيى1*، دجان بشير طه 2**

[2،*1] قسم البرمجيات، كلية علوم الحاسوب والرياضيات، جامعة الموصل، الموصل ، العراق

**الخلاصة :**

لقد ازداد تعقيد البرامج بسبب تعقيد وصعوبة المتطلبات الاضافية التي تتم اثناء عملية تحديث البرامج او اضافة وظائف جديدة والذي سيؤدي بالنهاية الى التقليل من جودة البرنامج ككل ، يمكننا تعريف اعادة بناء البرامج بانها احدى العمليات المضمنة في مرحلة الصيانة ضمن دورة حياة البرنامج وهي تقنية لتنظيف كود البرنامج من روائح الكود وتحسين البنية الداخلية للبرنامج وزيادة جودته باستخدام مجموعة من الفعاليات بدون تغيير السلوك الخارجي للبرنامج ، طور الباحثون تقنيات لإعادة هيكلة البرنامج على مستوى كود البرنامج او مستوى التصميم لتقليل الوقت والجهد اللازم اثناء اجراء عمليات الصيانة ، تضمنت هذه الورقة مراجعة منهجية للأدبيات لسبعة عشر دراسة على مستوى الكود ، ولقد اجرينا بحثاً آلياً في المكتبات الرقمية عن البحوث التي لها صلة بهذا الموضوع والمنشورة في الفترة الزمنية ما بين 2014 – 2021 وقمنا باختيار افضل خمسة باحثين في هذا المجال ولكل باحث تم اختيار خمسة

بحوث او اقل بالاعتماد على عدد مرات الاقتباس في قاعدة بيانات المواقع العلمية او استخدام منهج او طريقة جديدة والحصول على نتائج ممتازة، واخيراً قمنا بتحليل كل ورقة وذكر الطريقة او الخوارزمية المستخدمة في اعادة بناء البرنامج ، اهداف ونتائج كل ورقة .

**الكلمات المفتاحية :** اعادة بناء البرنامج ، تطوير البرنامج ، روائح الكود البرمجي

## 1. Introduction

The requirement of maintenance to evolve is considered an inherent aspect regarding software in real-world settings. The code is going to be more sophisticated and drifts away from the original design when software is expanded , adapted, and adjusted to new requirements, reducing the product's quality. As a result, software maintenance consumes the majority of total software development costs [1]. Better tools and processes of the software's development do not alleviate the problem since their enhanced capacity is utilized for implementing more new requirements in the same time frame, increasing software's complexity. For handling such a spiral of complexity, solutions that minimize software complexity through incrementally increasing internal software quality are urgently needed. Refactoring is the study domain which that tackles such topic [2,3].

"A change made to the internal structure related to the software to make it cheaper to alter and simpler to comprehend without modifying its observable behavior," according to the definition [4]. Refactoring improves the process of software development through by making programs more comprehensible, making software bugs easier to identify, and speeding up development. Refactoring, according to numerous researchers, could take place at the design, code, or requirement levels [5][6] .

In literature, various automatic refactoring methods were proposed for assisting practitioners in spotting smelled code and proposing refactoring procedures for correcting them. Those methods depend on rules, data driven , machine learning, or software space in order to maximize predetermined metrics. All of such methods have advantages and disadvantages. Rule-based approaches, for instance, produce excellent outcomes, yet stating the rules regarding a bad smell is not often straightforward. Users normally define the rules manually, which is a complex and time consuming operation.

Authors may use a fully or semi-automatic algorithms to find a refactoring opportunity (also referred to as a design flaw or a bad smell). Semi-automatic algorithms allow users (usually experts) to examine the algorithm's decision before it is applied, while fully automated techniques implement the refactoring decision immediately to the software artifact. In the case when multiple scientific databases have been searched, over 4000 published research discussing automatic refactoring at the design or code levels were found in the present work. Those studies have been published between the early 1990s and the year 2022. A total of 17 high quality publications relevant to automatic refactoring have been considered for this current work. Those scientific publications were chosen for their suggested novel approaches concering with regard to automatic refactoring, as well as their excellent outcomes and citations in the scientific sites database.

## 2. Related work

Many reviews of refactoring processes were undertaken as part of this study. [7] looked at 47 publications and found 47 refactoring opportunities. The researcher thought about refactoring opportunities, methods for discovering refactoring opportunities, empirical techniques for validating identification approaches, and dataset types. The researcher concluded that many of studies on refactoring opportunities concentrated on small datasets and ignored huge industrial datasets. Between 2009 and 2013, the number of research tackling refactoring opportunities increased dramatically , In contrast to those researches, the researchers analyzed the refactoring procedures themselves, whereas the current work concentrates on the methodologies utilized to accomplish such operations in this research. [8] expanded on [7]'s SLR by looking at refactoring and code smells, with a focus upon on identifying code smells and anti-patterns. The study uncovered datasets and technologies that have been utilized in the research that were detected. They indicated the most commonly utilized bad

smells in experiments. [9] gave an outline regarding the software refactoring and restructuring field. They covered the methods utilized for implementing refactoring, refactoring dependencies, refactoring scalability, and refactoring applications at high abstraction levels. [10], conducted a systematic review regarding the works for proposing, implementing, or suggesting an automated refactoring approach. The above-mentioned researches mentioned are primarily concerned with locating studies connected to highly particular or specialist refactoring subjects and sub-areas.

## 3. TYPES OF SOFTWARE ARTIFACTS

Refactoring could be utilized for any software artifact's type, event though most IDEs just support it for source code. Refactoring design models, software architectures, database schemas, and software requirements, for instance, is useful and possible. Refactoring such types of software artifacts relieves developers regarding various implementation-specific details while also increasing the changes' expressive power.

1- **Program**: In various paradigms and programming languages, support for program refactoring and restructuring was offered. Since control flow and data flow are extremely interwoven, it is more complicated to restructure programs that are not designed in object-oriented language. As a result, restructurings are usually restricted to the code block or function level [11]. It's also worth noting that the more complicated the language is, the more complicated it is to automate the refactoring. It is a popular software development technique for upgrading a software system's internal code structure without changing its external behaviors. Comprehending the way that the developers refactor source code might help to understand the process of software development and how different system versions interact. Different prominent programming languages, like Python and Java, offer refactoring detection tools [12] .

2- **Designs**: Refactoring at the level of the design, for instance in a UML model form, is a contemporary research topic. State chart diagrams, class diagrams, and activity diagrams can all be refactored using those concepts. The user might utilize refactoring to each of the diagrams that aren't simply or intuitively described in other source codes or diagrams [13].

3- **Software Requirements Restructuring** : This concept could be utilized as well to requirements specifications , [14] proposes restructuring natural language requirements conditions by dividing them into viewpoint structures. Each one of the viewpoints embodies certain system components' partial requirements, and interconnections between viewpoints have been made explicit. Furthermore, this restructuring method improves understanding of requirements and makes finding inconsistencies and managing requirements evolution easier.

## 4. Research Methodology

The review methodology includes establishing the databases to be searched, the development of research questions, data analysis, data collection ,and discussion of findings. The systematic literature review's goal is to uncover gaps that have been discovered by researchers before to the survey.

Finalizing databases to be searched, designing research questions, and analyzing findings and discussions are all part of the review methodology. The procedure entails searching supplementary and primary databases, applying exclusion and inclusion criteria, closing with discussions, and generating results. Refactoring has received contributions from more than 5584 authors worldwide [15]. In this paper to emphasis on four active authors, with just five researchers chosen for each depending on the number of citations and publications in the refactoring field area, as shown in Table (1) and Figure (1) . For making the search as broad as possible, we searched all of the adopted researche in the work using a variety of scientific literature sources such as the IEEE Xplore, ACM Library, and Academia.

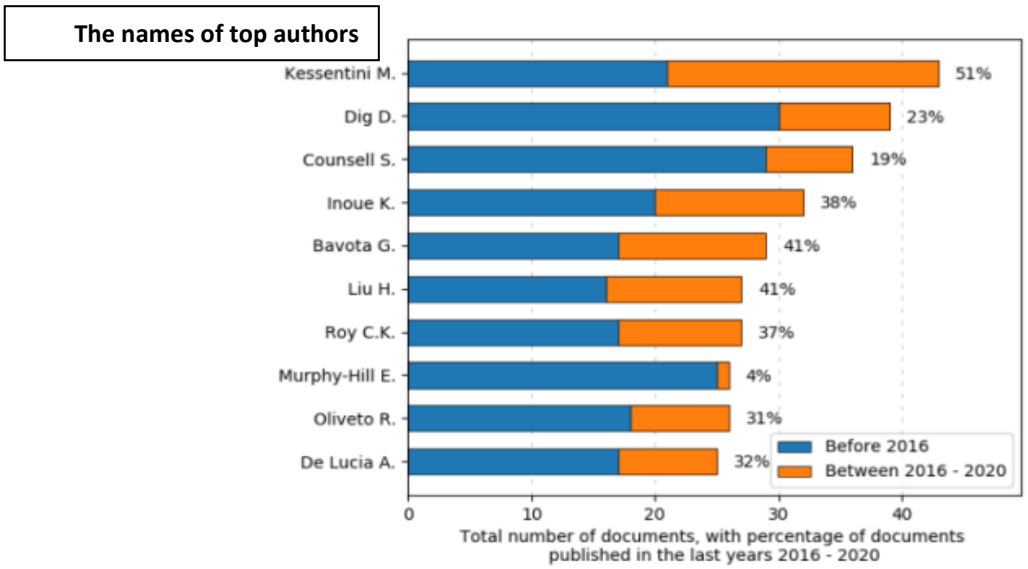


Figure 1. Top 10 Authors with a maximum number of citations and publications in the refactoring area [15]

The study selection process was divided into two stages: pre-selection and selection. The four adopted authors' names were used to screen studies during pre-selection. As a result , a 40 types of researches on the topic of refactoring have been published. In the second stage, criteria for exclusion and inclusion are established, with the inclusion criteria requiring that all researches be published in a conference paper or journal article between 2014 and 2021 and be retrieved from one of five databases, as well as a comparison depending on the citation numbers of the 40 studies. Exclusion criteria have been used for eliminating researches that was not relevant to the software refactoring, as well as researches written prior to the year 2000. We eliminated 23 researches as a result of this filtering process, leaving only 17 researche to be considered research that matched the inclusion requirements.

Table 1: Shows the studies that were be chosen

| Authors Name | Article | Digital Libraries | Citation No. | Year |
|---|---|---|---|---|
| Marouane Kessentini | [16] | ACM Library | 108 | 2014 |
| Danny Dig | [17] | ACM Library | 73 | 2016 |
| Marouane Kessentini | [18] | ACM Library | 125 | 2016 |
| Gabriele Bavota | [19] | IEEE Xplore | 36 | 2016 |
| Marouane Kessentini | [20] | Springer | 62 | 2017 |
| Gabriele Bavota | [21] | IEEE Xplore | 167 | 2017 |
| Roy ck | [22] | IEEE Xplore | 28 | 2017 |
| Danny Dig | [23] | IEEE Xplore | 206 | 2018 |
| Marouane Kessentini | [24] | IEEE Xplore | 32 | 2018 |
| Gabriele Bavota | [25] | Springer | 216 | 2018 |
| Gabriele Bavota | [26] | IEEE Xplore | 16 | 2018 |
| Roy ck | [27] | IEEE Xplore | 23 | 2018 |
| Roy ck | [28] | IEEE Xplore | 12 | 2019 |
| Roy ck | [29] | IEEE Xplore | 13 | 2019 |
| Roy ck | [30] | IEEE Xplore | 12 | 2020 |
| Marouane Kessentini | [31] | Elsevier | 20 | 2021 |
| Gabriele Bavota | [32] | IEEE Xplore | 1 | 2021 |

## 5. Literature Review of Refactoring

This section summarizes the chosen researches, which are organized based on the author's name and the number of citations (as shown in Table 1)

In 2014 , Kessentini et al proposes a distributed optimization problem for detecting code smells. Throughout the process of optimization, various approaches are integrated in parallel in order to reach a consensus on code smell detection. employed Parallel Evolutionary Algorithms (P-EA), in which numerous evolutionary algorithms with diverse adaptations (solution representations, fitness functions, and change operators) have been conducted in parallel cooperatively in order to accomplish a common aim of detecting code smells. An empirical comparison regarding the implementation of this cooperative P-EA technique with random search, 2 single population based methods, and 2 non meta heuristics search code smells detection approaches. According to a benchmark of 9 large open source systems with over 85% recall and precision scores on 8 types of code smells, statistical analysis regarding the acquired results offers evidence for supporting the claim that cooperative P-EA is more effective compared to the state of the art techniques of detection. [16]

In 2016 , Lin et al. , defined the Refactoring Navigator as one of the tool supported and interactive recommendation methods for architectural refactoring, according to the researchers. This method starts with a given implementation and finishes with a desired high level design, iteratively recommending a sequence of refactoring procedures. Furthermore, this method provides the users with the ability to accept, rejecting, and ignoring a refactoring step recommendation, and it incorporates the user's comments into additional refactoring recommendations. They used an industrial case studies and a controlled experimentation in order to assess the efficiency of their tool and method. [17] . In the sameyear, Ouni et al. , suggested multi objective search based method for automatic refactoring recommendations. The goal of this method is to discover the optimal refactoring sequence which (i) enhances quality through reducing the number of design defects, (ii) reduces the number of code changes needed to repair these defects, (ii) maximizes consistency with previous code modifications and (iv) retains design semantics. The efficiency of the proposed methods was assessed with the use of empirical sizing regarding 6 open-source systems .[18]. In the sameyear , Anich et al. interviewed and surveyed 53 Model-View-Controller pattern (MVC) developers. After that, they used an open coding approach to create a list of 6 Web MVC smells, including Fat Repository, Brain Repository, Brain Controller , Promiscuous , Controller , Meddling Service and Laborious Repository Method. They then conducted research on 100 MVC projects to see how much smells affected code change and problem proneness. The findings reveal that Web MVC smells (a) frequently raise the change and defect proneness of classes, (b) are viewed as severe issues by developers.[19]

In 2017 , Mansoor et al. proposes a new system that allows software designers to perform the model level refactoring. They achieved this by employing a multi objective evolutionary algorithm in order to establish a balance between enhancing the class quality and diagrams of activity. In addition, the suggested multi objective method offers software designers a multi view to analyze the influence of the proposed refactoring applied to the class diagrams on related diagrams of activity for assessing overall quality, ensuring behavior preservation, and validating feasibility.[20] , at the same year , Tufano et al. looked at various studies that indicated code smells had a negative effect on code maintainability and comprehensibility. Whereas the effects of smells on code quality were objectively evaluated, there’s still just anecdotal data on why and when bad smells are presented, how long they last, and how developers remove them. They carried out a major empirical analysis of the change history regarding 200 open source projects to empirically validate this anecdotal data. This research necessitated the creation of an approach for detecting smell introducing changes, as well as mining of more than half a million commits and manual analysis and classification of over 10,000 of them. Their results largely contradict popular belief, demonstrating the fact that the majority of small instances occur during the creation of an artifact rather than as a result of its evolution [21] , also in the same year Mondal et al. looked into which clone fragments will possibly have bugs, so that they could be prioritized for tracking and refactoring in order to reduce future bug fixing tasks. Current research on clone bug proneness is unable to identify code clones that will potentially receive future bug fixes. Change frequency regarding code clones doesn't imply their bug proneness, according to their examination of thousands of revisions of 4 different subject systems that have been developed

in Java . Bug proneness has been primarily linked to the frequency with which code clones are changed. As a result they discovered that the bug proneness of code clones is mostly determined by how lately they were altered or produced . They feel that code clones must be given priority by management because of their recent creation or alteration . [22]

In 2018 , Tsantalis et al. study created, tested, and assessed RMiner, an approach that addresses the restrictions mentioned above. An AST-based statement matching method is at the heart of RMiner, which finds refactoring candidates without the need for user defined thresholds. To test RMiner, they built the most extensive oracle to date, which leverages triangulation to produce a dataset with far less bias, containing 3,188 refactoring from 185 open-source projects [23] , at the same year Alizadeh et al. proposed an interactive method for reducing the developer's interaction efforts in the case when modifying systems by integrating multi objective and unsupervised learning. They develop multiple feasible refactoring solutions by identifying a balance between many competing quality attributes with the use of multi objective search. After that, using an unsupervised learning method known as Pareto front, the developers are guided in identifying their region of interest and reducing the number of refactoring choices to investigate. The developer's feedback is utilized to automatically establish constraints for reducing the search space in following iterations and concentrate on the developer's preferred region. They chose 14 active developers in order to manually test our tool's efficacy on one industrial system and five open-source projects. The findings revealed that participants discovered their intended refactoring faster and more precisely compared to the existing state of the art. [24] . In the sameyear spite of the research community's efforts to understand code smells, the extent to which the code smells in software systems affect the maintainability of the software is unknown. Palomba *et al*. And others report on a large scale empirical research of code smells' diffuseness and effect on a code change and fault proneness. The research looked at 395 releases from 30 open source projects and 17350 manually validated instances of 13 types of code smells. The findings reveal that smells associated with complex and/or long code (Complex Class) are widely dispersed, and that smelly classes are more prone to modification and defect compared to smell free classes. [25] . In the same year Pantiuchina et al. , according to the researcher, empirical investigations have shown that bad code quality is often related to lower maintainability. As a result, technologies to automatically discover design flaws (code smells) were developed. These technologies, on the other hand, are unable to prevent design flaws from being introduced. This means that before state of the art technologies can be used to uncover and repair design flaws, the code must degrade in quality, to prevent the introduction of design flaws through refactoring operations instead of addressing them after they have already affected the system. They call this new approach to software refactoring "just in time refactoring." Furthermore, they took a first step in this direction by giving a method for predicting which classes may be impacted through code smells in future.[26] . In the same year Mondal et al. , looked into the role of micro clones (code clones with no more than five lines of code) in maintaining a consistent code base. While prior clone detectors and trackers have overlooked micro clones, their analysis of thousands of commits from 6 subject systems suggests that micro clones account . Depending on statistical significance analyses, the percentage of consistent updates in micro clones is much higher in comparison with the ones in the regular clones. according to their manual study micro-clones, such as regular clones, need consistent updates, and tracking or refactoring micro clones could significantly reduce the efforts required to keep them up to date. [27]

The majority of previous researches on code clones overlooked micro-clones, which might range in their size from 1 to 4 LOC. In 2019 Islam et al. compares the bug-proneness regarding micro-clones with conventional code clones in the year 2019. They discover and study regular as well as the micro-clones which are connected to reported issues from hundreds of revisions of 6 different open-source subject systems built in 3 programming languages (C, C#, and Java). Micro-clones had a considerably larger rate of changed code fragments owing to bug-fix commits compared to regular clones, according to their research. Micro-clones a substantially higher number of consistent modifications owing to bug-fix commits in comparison with the regular clones. Also, they discovered that bug-fix

commits influence a considerably higher percentage of files in micro-clones compared to regular clones. Lastly, they discovered that the percentage of severe bugs in micro-clones is substantially higher in comparison with the regular clones. They used Mann-Whitney-Wilcoxon (MWW) test in order to determine the statistical significance of our findings. Furthermore, their results suggested that when it comes to software maintenance and clone management, micro-clones must be prioritized. [28] . In the same year Mondal et al. , Context Adaptation Bugs, or simply Context-Bugs, are the term used by the authors to describe bugs that could be found in code clones. They analyzed and defined two clone evolutionary patterns which show Context-Bugs are being fixed. Code cloning frequently causes Context-Bugs in software systems, based on their examination of thousands of modifications of 6 open source subject systems developed in C, Java, and C#. Context-Bugs account for almost half of all clone-related bug fixes. Cloning (copy/pasting) a newly-created code fragment (in other words, one not introduced in a previous revision) is more possible to introduce Context-Bugs than cloning an existing fragment (a code fragment that has been added in a former revision). Furthermore, when cloning across distinct files, the likelihood of generating Context-Bugs appears to be substantially higher than when cloning within the same file. Lastly, among the 3 major clone-types, Type 3 clones (i.e. gapped clones) have the maximum likelihood of containing ContextBugs. Their results could be useful in detecting and removing Context-Bugs in code clones early on.[29]

In 2020, Mondal et al. , presented a survey of current clone refactoring and tracking approaches, as well as future research opportunities in such fields. They specified the features of the quality assessment for clone refactoring and tracking tools, then compared such features among the tools. Their survey is the first to look into clone re-factoring and tracking in depth. Based on their clone refactoring survey, automatic refactoring cannot eliminate the need for manual attempts in identifying refactoring opportunities and testing system behavior after refactoring. Quality assurance engineers may have to devote a considerable amount of effort and time to post-refactoring testing. The impact of clone refactoring on system performance has received little attention. They also believe that more research into real-time detection and tracking of the code clones in big-data environment is required in the future. [30]

in 2021 AlOmar et al. , intended for better understand what motivates the developers to use refactoring through mining and automatically classifying a large sample of 111884 commits comprising refactoring activity that has been extracted from 800 open source Java projects. Along with standard Bug Fix and Functional categories, they have trained multiclass classifiers for categorizing such commits into 3 categories: External Quality Attribute, Internal Quality Attribute, and Code Smell Resolution. The conventional definition of refactoring, which limited it to enhancing software design and eliminating code smells, is challenged by this classification. They qualitatively studied commit messages for extracting textual patterns which developers commonly employ to explain their refactoring operations in order to better understand their classification results. Their findings reveal that (1) eliminating code smells isn't the primary motivation for developers to modify their code bases. Beyond its usual definition, refactoring is requested for various reasons; (2) Refactoring operations are distributed differently in production and testing files; (3) developers utilize a range of patterns to target re-factoring-associated activities; and (4) textual patterns that have been obtained from commit messages give better coverage for the way that the developers describe their re-factorings. [31]

In the same year Traini et al. , attempted to close such gap by presenting the largest study till now on the influence of refactoring on the on software performance with regard to execution time. They examined the modification history of 20 systems that had performance benchmarks established in their repositories in order to find commits where developers performed refactoring operations on code components that the performance benchmarks exercised. They demonstrated that refactoring operations might have a considerable influence on execution time via a qualitative and quantitative examination. Actually, none of the refactoring forms explored may be deemed "safe" in terms of preventing performance regression. Extract Method, Extract Class/Interface, and other

refactoring types aimed at the decomposition of complex code entities have higher risks of causing performance degradation, indicating that they have to be considered carefully in the case of modifying performance-critical code. [32]

## 6. Discussion

Various empirical research investigated the role of refactoring approaches without identifying the different types of refactoring approaches. Statistically, eight out of seventeen research (or 46% of the total) found no refactoring methods ( [25] , [26] , [27] , [28] , [29] , [30] , [31] , [32] ). However, as indicated in Tables 2, nine out of seventeen studies (or53%) have identified types of refactoring approaches used in their research.

**Table 2: Refactoring Methods utilized via researchers**

| Author | Study | Refactoring Methods or Refactoring algorithm |
|---|---|---|
| *Marouane Kessentini* | [16] | Parallel Evolutionary algorithms (P-EA) |
| *Danny Dig* | [17] | Refactoring Navigator Method |
| *Marouane Kessentini* | [18] | A benchmark of 6 open-source systems, move field, move method, pull up field, push down field, pull up approach, push down approach, move class, extract approach, inline class, extract class and extract interface |
| *Gabriele Bavota* | [19] | Parallel Evolutionary algorithms (P-EA) |
| *Marouane Kessentini* | [20] | Multi-objective evolutionary algorithm |
| *Gabriele Bavota* | [21] | This study Reviewed some researches have shown the negative impacts of the code smells on code maintainability and comprehensibility |
| *Gabriele Bavota* | [22] | This research examines the clone evolution history regarding four Java-based subject systems over thousands of revisions, and examines how change-proneness of clones of the code is connected to their ability to contain bugs. |
| *Danny Dig* | [23] | Design, implement, and evaluate RMiner Method |
| *Marouane Kessentini* | [24] | Interactive method that combines utilization of the unsupervised and multi objective learning for the reduction of developer's interaction efforts when refactoring system |
| *Gabriele Bavota* | [25] | This study reported a large study that has been carried out on 395 releases of 30 Java open source projects, which have been targeted at the understanding of diffuseness of the code smells in Java open source projects and their relation with the source code change and fault proneness. The study had considered 17350 samples of 13 different types of the code smell, which has been first detected with the use of metric-based method and validated manually after that. |
| *Gabriele Bavota* | [26] | The code smell predictor was utilized in this work for predicting if a given class would be affected via specific sort of code smell within t days. Customize the code smell type to anticipate and the threshold of value t. |
| *Gabriele Bavota* | [27] | The significance of micro clones (code clones of 4 LOC or fewer) in software maintenance and evolution is investigated in this paper. Existing research has often overlooked such small clones, supposing that they are spurious clones. |
| *Gabriele Bavota* | [28] | This work looked at and compared bug proneness and properties of regular and micro-clones on six different topic systems, and discovered that microclones require more care throughout software maintenance than regular clones. |
| *Gabriele Bavota* | [29] | This research looked at context adaptation bugs (also known as context bugs) in 3 different code clone types. Context bugs are bugs that are introduced to clone fragments as a result of the fragments not being correctly adapted to their respective contexts. |
| *Gabriele Bavota* | [30] | This research presented a survey of current clone refactoring and tracking researches, methods, and tools. All of the researches are categorized according to their study directions, and they indicate how far each area was researched. Also, discover current clone refactoring and tracking tools and compare them according to their characteristics. |

| | | |
|---|---|---|
| *Marouane Kessentini* | [31] | Trained a multi class classifier for the categorization of refactoring activities into 3 categories, namely, External Quality Attribute, Internal Quality Attribute, and Code Smell Resolution, along with conventional Bug Fix and Functional categories |
| *Gabriele Bavota* | [32] | This research looked at all of the implications of various refactoring kinds on software performance. |

## 7. Conclusion

The results of the systematic review of software refactoring indicated that the identification of refactoring opportunities is a highly active area of research 17 researches were chosen as primary studies and thoroughly examined. We emphasized four top active writers, with just five studies or less chosen for each depending on the number of citations and publications in the refactoring field area .The goal of this review is to find out which refactoring methods were used in the studies that were used. Furthermore, numerous papers found no refactoring approaches, even though such articles are statistical investigations of refactoring procedures. There is little empirical research on the refactoring methods in use. The researchers concluded that the use of machine learning algorithms in refactoring method and detecting bad smells in the source code is the best and gives very accurate results despite the large database used in the training. Also, by using Semi-automatic algorithms allow users to examine algorithm's decision before it is applied To check whether this process increases the quality of the program or not.

**REFERENCES**


[1] N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies : A Systematic Mapping Study Electronic Research Archive of Blekinge Institute of Technology Citation for the published Journal paper : Title : In press : Software Development in Startup Companies : A Systematic M," *Inf. Softw. Technol.*, no. January, 2014.

[2] A. Barriga, L. Bettini, L. Iovino, A. Rutle, and R. Heldal, "Addressing the trade off between smells and quality when refactoring class diagrams," *J. Object Technol.*, vol. 20, no. 3, pp. 1–15, 2021, doi: 10.5381/JOT.2021.20.3.A1.

[3] G. Islam and T. Storer, "A case study of agile software development for safety-Critical systems projects," *Reliab. Eng. Syst. Saf.*, vol. 200, no. April, 2020, doi: 10.1016/j.ress.2020.106954.

[4] A. Shrivastava, "Martin Fowler - Refactoring Improving the Design of Existing Code".

[5] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "A survey on UML model smells detection techniques for software refactoring," *J. Softw. Evol. Process*, vol. 31, no. 3, pp. 1–17, 2019, doi: 10.1002/smr.2154.

[6] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004, doi: 10.1109/TSE.2004.1265817.

[7] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 231–249, 2015, doi: 10.1016/j.infsof.2014.08.002.

[8] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Eng. J.*, vol. 9, no. 4, pp. 2129–2151, 2018, doi: 10.1016/j.asej.2017.03.002.

[9] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - Improving coupling and cohesion of existing code," *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 144–151, 2004, doi: 10.1109/WCRE.2004.33.

[10] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Softw. Qual. J.*, vol. 28, no. 2, pp. 459–502, 2020, doi: 10.1007/s11219-019-09477-y.

[11] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," *Conf. Rec. Annu. ACM Symp. Princ. Program. Lang.*, no. July 2000, pp. 155–169, 2000, doi: 10.1145/325694.325713.

[12] H. Atwi *et al.*, "PYREF: Refactoring Detection in Python Projects," *Proc. - IEEE 21st Int. Work. Conf. Source Code Anal. Manip. SCAM 2021*, pp. 136–141, 2021, doi: 10.1109/SCAM52516.2021.00025.

[13] D. Arcelli, V. Cortellessa, and D. Di Pompeo, "Automating Performance Antipattern Detection and Software Refactoring in UML Models," *SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 639–643, 2019, doi: 10.1109/SANER.2019.8667967.

[14] M. Majthoub, M. H. Qutqui, and Y. Odeh, "Software Re-engineering: An Overview," *2018 8th Int. Conf. Comput. Sci. Inf. Technol. CSIT 2018*, no. July, pp. 266–270, 2018, doi: 10.1109/CSIT.2018.8486173.

[15] C. Abid, V. Alizadeh, M. Kessentini, T. do N. Ferreira, and D. Dig, "30 Years of Software Refactoring Research:A Systematic Literature Review," vol. 1, no. 1, pp. 1–23, 2020, [Online]. Available: http://arxiv.org/abs/2007.02194

[16] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection," *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 841–861, 2014, doi: 10.1109/TSE.2014.2331057.

[17] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. 13-18-Nove, pp. 535–546, 2016, doi: 10.1145/2950290.2950317.

[18] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, 2016, doi: 10.1145/2932631.

[19] M. Aniche, G. Bavota, C. Treude, A. Van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," *Proc. - 2016 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2016*, no. January 2021, pp. 233–243, 2017, doi: 10.1109/ICSME.2016.12.

[20] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Softw. Qual. J.*, vol. 25, no. 2, pp. 473–501, 2017, doi: 10.1007/s11219-015-9284-4.

[21] M. Tufano *et al.*, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, 2017, doi: 10.1109/TSE.2017.2653105.

[22] M. Mondal, C. K. Roy, and K. A. Schneider, "Identifying Code Clones Having High Possibilities of Containing Bugs," *IEEE Int. Conf. Progr. Compr.*, pp. 99–109, 2017, doi: 10.1109/ICPC.2017.31.

[23] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," pp. 483–494, 2018, doi: 10.1145/3180155.3180206.

[24] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," *ASE 2018 - Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, no. September 2018, pp. 464–474, 2018, doi: 10.1145/3238147.3238217.

[25] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, 2018, doi: 10.1007/s10664-017-9535-z.

[26] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk, "Towards just-in-time refactoring recommenders," *Proc. - Int. Conf. Softw. Eng.*, no. May, pp. 312–315, 2018, doi: 10.1145/3196321.3196365.

[27] M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," *25th IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2018 - Proc.*, vol. 2018-March, no. June 2020, pp. 50–60, 2018, doi: 10.1109/SANER.2018.8330196.

[28] J. F. Islam, M. Mondal, and C. K. Roy, "A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones," *SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, no. June 2020, pp. 73–83, 2019, doi: 10.1109/SANER.2019.8667993.

[29] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "Investigating Context Adaptation Bugs in Code Clones," *Proc. - 2019 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2019*, pp. 157–168, 2019, doi: 10.1109/ICSME.2019.00026.

[30] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," *J. Syst. Softw.*, vol. 159, no. June 2020, 2020, doi: 10.1016/j.jss.2019.110429.

[31] 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, no. October, 2021, doi: 10.1016/j.eswa.2020.114176.

[32] L. Traini *et al.*, "How Software Refactoring Impacts Execution Time," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–23, 2022, doi: 10.1145/3485136.