Hassan et al. LGURJCSIT 2024

LGU Research Journal of Computer Science & IT

ISSN: 2521-0122 (Online)
ISSN: 2519-7991 (Print)

doi: 10.54692/lgurjcsit.2024.082558

Vol (8): Issue (2), April – June 2024

Use of Code Refactoring Transformation in Software Advancement

Taimoor Hassan[1*], Abrar Ahmed[2], Mehmood Anwar[3], Shahzaib Afzal[4], Muhammad Mohsan[5], Muhammad Basit Ali Gilani[6]

[1]Department of Software Engineering, University of Central Punjab, Lahore, Pakistan.
[2,3]Department of Software Engineering, The University of Lahore, Lahore, Pakistan.
[4,5]Riphah Institute of Computing & Applied Sciences, Riphah International University, Islamabad, Pakistan.
[6]Department of Computer Science, University of Central Punjab, Lahore, Pakistan.

Email: taimoor.hassan01@ucp.edu.pk

ABSTRACT:

In this paper, the refactoring of Object-oriented code affects the software quality in some ways. Reengineering, which means refactoring, is nearly always a good idea and is usually cheap; it optimizes the structure where the behavior does not appear to the user to have changed. This advantage makes it a subject of much research when as a tool for measuring the quality of software. It is proposed that software requirements be addressed at this stage since rather specific requirements are associated with higher-quality work. Refactoring is cost-effective when done during the development phase, this is because requirements that may need refactoring are identified early. It is necessary to note that our paper consists of five research questions and their answers. Some of the best practices followed in case of requirements management include Collection of requirements, choosing relevant requirements from the pool, further subversion of the selected requirements, placing the requirements in order of priority, and numbering and documentation of the requirements. This methodology is then implemented and results are attained at each phase of the study in the case of the Hotel Management System.

KEYWORDS: Refactoring, Transformation, Primary Studies, Systematic Mapping Study, Structured Query Language.

1. INTRODUCTION

In the eyes of the spectators, the plot of the topic indicates the significant role of code refactoring in quality assurance, growth in agility, and ease in maintenance and sustainability. With the expansions of software systems complexity, it is expected that the implementation of code refactoring as a regular practice also increases, which would improve speed and the spirit of innovation.

Refactoring as a process control concept has been a component of software engineering for decades. Originally, it was introduced by Martin Fowler and it refers to the process of refactoring the code that already exists to enhance its internal organization. This has become important especially in agile, which is characterized by frequent cycles and improvement. Moreover, Code refactoring is the process by which an existing code is changed for the better in terms of quality standards but the external interface remains the same. It is a way to decrease the technical debt, make updates less complex, and guarantee the software's further maintainability. Essentially, the main reason to refactor is the dynamic nature of the requirements and technologies in use. Another disadvantage of using software systems is that as the complexity

of the systems increases the management also becomes a herculean task. Refactoring facilitates the enhancement of the source code system in various ways such as; enhancing flexibility, reducing the number of defects, and increasing the capacity of production among developers. Furthermore, the problems of refactoring are usually long and can make creating features take significantly longer in the short term. Modifications to the code regarding its structure might create new problems in the program. It is relatively easier for stakeholders to get a handle on why the refactoring has to be done as distinct from seeing feature addition benefits.

Furthermore, this research aims to execute inclusive methodical mapping research on previous experimental studies to evaluate the consequence of code refactoring activities for object-oriented to measure the quality attributes of the software.

Software maintenance is one of the exclusive and intense struggles for activities of development of software [1] [2]. Software maintenance higher in cost shows the bad design quality of software [3]. In the maintenance phase of software, several code modules are by mistake presented.

the developers [4]. These modules of code or parts of code show the bad quality of software at a later time due to several code changes [5] [6]. In this research, the author identified 13,283 articles relevant to the study from six digital libraries. Later, they scrutinized research that is based on several viewpoints, which include abstract, title, and full text, using searching manually and by reference. Further, they analyze the quality of the study by selecting 142 Primary Studies (PSs). They classify these primary studies on behalf of the refactoring activities influences. The author also discusses the numerous refactoring activities such as quality measures of software, statistical practices, datasets, and quality attributes, working using nominated PSs. Further, the author extracts the testified software tools that forecast or evaluate the influence of refactoring actions. They also provide current and previous studies on the outcome of code refactoring methods of object-oriented to enhance the attributes of the software quality.

In conclusion, they organize the distributed and inconsistent discoveries to determine a list of exposed problems and challenges that are required to be addressed in the future. The research problem which is addressed in this research is that author extracts a clear picture by constructing systematic research that analyzes and reports the conclusions on the association amid refactoring methods and the enhancement of quality for object-oriented applications or software. This paper is divided into different sections, section II explains the related work, III section elaborates on the research methods, section IV defines the problem statement, V section explains the proposed solution, section VI shows the results, VII section explores the discussion, and the last, summarize the overall paper in conclusion section VIII.

2. LITERATURE REVIEW

Code refactoring impacts the software quality, performance, and maintainability deeply, adding the crucial components of a successful outcome. It has a multitude of benefits such as better readability, increase in performance, and flexibility but at the same time, it has its shortcomings, i.e. introducing new bugs from other resource consumptions. Hence, we need to come up with a well-designed plan, communicate with the team, and test the refactoring result before we decide to move it to the main codebase [7].

Furthermore, the objective for recovery-focused is just to enhance the level of technology as well as the competitiveness of developers. Over the previous two decades, several longitudinal experiments were performed to examine the effect of recompilation operations on technology efficiency. The objective of this project [8] would be to conduct an expected to be stable vulnerability assessment of current academic research on the role of expression application regression testing practices on application performance parameters. According to the findings, scientific work observed a greater detrimental effect of recompilation with application efficiency whereas research was published across sectors. Except for consistency, difficulty, succession, mistake propensity, and thermal dissipation, both consistency parameters improved rather than degraded because of rewriting practices. Besides that, many performance indicators investigated throughout research papers provide dynamic influence on individual requirements engineering operations, meaning whether modification often does not increase certain performance characteristics. The whole research identifies the list of available problems that need to be examined more deeply, such as a scarcity of technical verification, the restricted scope with provocation, and minimal device

funding, just to name a few. Moreover, the study consists of a comprehensive research methodology analysis that describes, evaluates, and summarizes its previous research evidence on the effect of rewriting practices to graphical fidelity, only to evaluate its latest developments while identifying new research directions. To begin, they presented an overview of both the programmer requirements engineering system. They then spoke about the visualization study methodology that was used to perform that applicable study contributes to certain phenomena under investigation are searched using two techniques [9].

On the other hand, they started by searching certain internet sources: Embedded Celia de, Cochrane Library, Walther, Research gate, PubMed, including Greene. Then, in all the five most important famed publications, a physical review was undertaken. We have used comparison-losing games to improve the accuracy of their quest but reduce the chance of losing important information. The distribution of PSs by presentation month revealed that assessing the impact of recompilation practices for system development has become a hot topic throughout academia. Those findings suggest also that the Transfer System and extraction Process, including Extraction Group requirements engineering practices, has received more attention from investigators than many of the other requirements engineering operations [10].

Meanwhile, scholars have contributed less time and effort towards designing machine learning that can forecast their advantages from programmed regression testing until it is implemented. The impact of recompilation practices on user-friendliness was measured and found using a variety of various shaped samples applied across many instrument computer languages. Since only a few other Rag used computational methods to investigate the importance of expected returns, they used a cast-a-ballot method to combine their results more about the impact of recompilation with computer efficiency. This same effect from cumulative regression testing practices for various system integration indicators provides contradictory data, indicating that computer modification often does not boost certain application performance objectives. Likewise, various configuration operations have an opposing influence on global device consistency parameters. Eventually, they discovered how analyzing the impact of recompilation over graphical fidelity involves several measurements including the order in which provocation is applied, its user experience metric, and so forth. Every element's variance may influence this report's ultimate result. As a result, when planning the analysis, that element ought to be seriously evaluated [11].

However, the modification would be a popular method of improving software efficiency. The effect on something like a wide variety of computer luxury homes, such as reproducibility, ease of maintenance, and efficiency, was already thoroughly investigated but quantified. They evaluate the effect of some of the more standard programming change management guidelines on privacy laws inside this paper, utilizing protection indicators that can measure safety first from the perspective of future knowledge transfer [12]. Those statistics were estimated with each SQL statement that used a fixed methodology we created to examine extracted Separator dynamically. They used their Java programming language analyzer on several applications that have been remediated by the rules. Which pseudo-code systems' updated indicators data suggest that perhaps the programming updates used to have a meaningful impact on data protection.

Integration testing, [13] the method of enhancing the current program's functionality by modifying its internal configuration whilst altering any dynamic behavior, is often used to increase computer consistency through optimizing semantic layout, usability, and error reduction. This relates to the reorganization of the admin tool (qualities and both techniques) within other categories. Modifying would be a common practice of project implementation. For certain web applications, such as Requirements Elicitation, patching is considered the most important aspect of both the agile methodology [25]. Adjustments may be needed to maintain the computer's reliability. Responding to rising standards, technology keeps evolving throughout its lifespan. The code becomes dysfunctional because of such a development. Reduced technology could have highly amplified transformations of both its architecture. As a result, its expense of commitment to repairs would have been greater. As a result, there may be a critical requirement for reliable applications. Only at the right points, which reflect that greater degree of project management, unacceptable uncertainty may occur. At such a reduced scale, namely the grade level, unnecessary volatility will occur. The fundamental feature regarding product design through

out Expression applications was its category. Category consistency refers to how easily a computer object may develop whilst maintaining that structure, whereas technology stability refers to how adaptable code would be to changing requirements or environments even when maintaining that structure. Computer structures that have been well ought to be able to adapt while requiring significant appropriate adjustments [14].

Since structural modifications become complex to maintain, code should always be built keeping enhanced structural consistency considered in mind. Since groups are indeed an important part of the development infrastructure, programmers must continue to put them in as soon as reasonably practicable. Some criteria for the category of design stabilization were identified. Modifying technology requires a considerable amount of funds. People argue which source code becomes important because it increases programmer efficiency throughout. That requirement engineering process, on the other hand, seems to have a unique effect on user-friendliness. Computer programmers normally create for specific architectural purposes that might or might not be mutually exclusive [15].

As a result, programmers must aim to create robust applications in terms of improving the development process for key calligraphers. There have been no rules for computer programmers that the recompilation techniques to be used in time to retain code security. As a result, they must investigate the impact of recompilation approaches against computer stabilization but classify such source code approaches according to that observable impact upon system instability. Its category becomes aimed at helping computer developers through selecting suitable configuration strategies for maintaining consistent code [12]. The purpose of this article would be to evaluate the effect of recompilation upon category but structure stabilization, as well as to suggest a grading system among rewriting approaches that depend highly upon category but structure stabilization. It will also inform computer programming on whose recompilation techniques are being used to preserve code stability [16].

That benefit of responsive design in terms of readability would be much less apparent inside the brief period that can sometimes be believed, based on the circumstances. In this analysis, they examine how "spotless programming" source code improves programmer efficiency inside a technology platform including data structures inside an educational factory [16]. Significant rises or declines in intelligibility were found, demonstrating that rapid rises in intelligibility are not necessarily apparent. According to their findings, adapting software can lead together in a usability hit within a brief period if indeed the mechanical reinforcement diverges according to what programmers were becoming accustomed to. Huge, lengthy software applications sometimes tackle the issue of technological complexity, which is exacerbated by implementation methods that end in difficult-to-maintain coding [17]. The technological liability may be justified; furthermore, unless the liability also is not forgiven, the cost is always charged, who serves the purpose with extra resources duration by designers either neither learn nor modify complicated programming. Thus, according to Bennett, "that budget deficit of just an undistributed deployment will bring whole organizations to a halt. "Modification is indeed a system software reorganization strategy that could be used to pay back intellectual liability. Reuse is said to boost programmer morale, establish principles architecture, render applications easy to learn, and even assist programmers with finding glitches.

Matthews also makes the case that the composition of software affects high flexibility, but they encourage programmers to stay aware of certain software but write native apps. Even so, it has already been acknowledged that only some research has quantified its reported configuration benefits, significantly about production efficiency. When they evaluate its alleged configuration benefits from just one arm as well as the limited scientific data from the other, there is a lack of information [8]. If additional innovations were applied to both frameworks, several results show how modification will result in improved efficiency for many days, and therefore programmers who do also get a stronger copy of the concept [21]. That ongoing creation of the micro data center also led to significant corporate expansion, including software developers spread over several regions. The program's exponential expansion, and the total number of employees upon this, also resulted in technological borrowing.

3. RESEARCH METHODS

In the research method step, they followed a multi-stage shortlisting process for the selection

of 142 published research articles by December 2017. Proposed research questions are answered by different aspects using the classification of selected research articles. The vote-counting method is applied to syndicate the results and report the analysis in PSs. The proposed research questions for this study are the following:

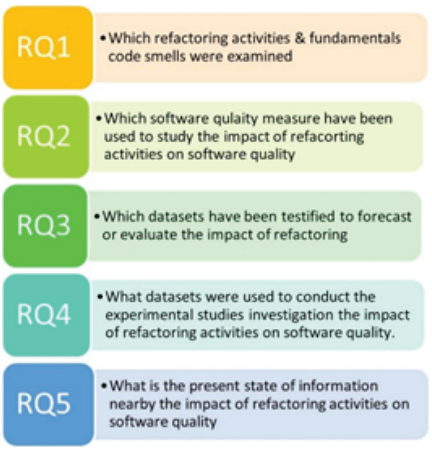

Figure 1: Research Questions

Several electronic databases are used in this study, some of these are the following.

Firstly, they draw major search terms from research questions, later in this research article the author obtains synonyms, abbreviations list, alternative spells, and core research terms. In the final section, they combine the core search terms using Boolean operators. For validation of the initial search, the string author selects 20 research articles from databases that contain the most relevant research articles. The search string is constructed which needs a search for titles, abstracts, and keywords of research articles. For this purpose, they selected 20 research articles, and only 11 out of 20 research articles were selected by their initial string search. Different types of searches are used by authors. These are the following:

3.1. Automatic search

In this step, they execute search strings of the articles in six electronic databases. In the first automation search, 12,996 search results were found. In the second phase, they found 287 more articles, where the total number of search results was updated to 13,283. These searches were organized and managed using Zotero.

3.2. Manual search

For completeness of the articles list, they perform a manual search analysis. This search results in 174 total articles in two phases, 161 and 13 search results, respectively. Manual search results were recorded using MS Excel spreadsheets after the exclusion of unrelated articles. After this search, both results were combined to eliminate the duplicate search results.

3.3. Reference Verification

They independently perform references using the manual examination of the reference list for the 129 applicable gained research articles.

The author set the Inclusion and exclusion standards which were useful for searching the articles that were not significant or related to core work concerning research questions. After phase 1, both authors independently verify this criterion using a set of 90 research articles arbitrarily from automated examination results. After this, they applied Cohen Kappa static [18] on another set of 90 research articles randomly. The author uses 7 stages to screen the articles, at stage 1, the author performs automatic and manual searches from six electronic databases. Search results were captured automatically using Zoterol. In stage 2, the author performs data cleaning of automatic exploration outcomes and separates unrelated entries. At stage 3, the author merges the outcomes of in cooperation automatic and manual explorations in MS Excel spreadsheets.

At stage 4th, the authors rejected research articles based on the article's titles. This process selects 1374 and 52 studies in Phase 1 and Phase 2 individually for further processing in step 5th. In the 5th stage, they screened both authors autonomously on behalf of the abstracts. In stage 6th both authors studied the complete manuscript of 247 and 37 research articles selected in both phases correspondingly and performed an inclusion and exclusion process on articles.

In the 7th stage, they verify references of selected articles. Results from this step were additionally distinguished on the behalf of abstracts and full texts. Finally, 142 primary studies were involved in the concluding list, and they extracted the relevant data from these selected research articles. Information obtained from a reading of these articles was documented in a data abstraction form. Items that are extracted from the individual primary study are the following. Full references from the PSs with title, author, publication title, and year. For study classification, the author uses 4 phase approach for the classification of selected PS into 7 facets which are research involvement

technique, study framework, refactoring activities, quality measures of the software, search method, focus, and datasets.

4. PROBLEM STATEMENT

Numerous software initiatives advance daily in our sector of the software business. Given the size of the investment in our nation, that is advantageous for the industry. These days, practically every firm uses a distinct software system to carry out daily office operations and activities. Thus, it is safe to state that not using and installing software systems in any kind of organization or business leads to a great deal of issues. The nation's largest investor and contributor to the growth of our businesses is the software sector.

As time goes on, the organization encounters a growing number of software requirements-related issues when developing software systems. It can be very challenging to grasp requirements when they are communicated by clients in their natural language. Clients without literacy are the primary problem with this. As a result, individuals fail to clearly state and articulate their needs about the suggested system. These clients can be found all over the place [19] to produce software from software companies. Because of the client's ambiguous criteria, the suggested program will not be excellent in every way in the end. The software development process is prone to numerous fluctuations due to the significant communication and conversation gaps that exist between the project team and the client. The client's lack of literacy and improper clearance of the requirements are the primary causes of this.

5. PROPOSED SOLUTION

It is very important to exemplify the applicability of the proposed approach as well as to prove its efficiency for practical usage, so we reveal the application and evaluation of the proposed approach based on the case study of the Hotel Management System. That is why in each stage of the development when applying our approach, we obtain tangible outcomes that describe the effectiveness of requirements-driven strategies with refactoring. The given case can be considered as a pilot for the demonstration of the results that can be achieved by implementing suggested strategies as the improvement of software quality and enhancement of development productivity are evident. Moreover, the suggested remedy for a particular issue is discussed in section IV of this section. Figure 2 below provides an overview of the suggested solution, detailing how we will create a software system that is error-free and intelligible to both the customer and the company's end user.

Figure 2: Used Approach

5.1. Requirement Gathering

Gathering the requirements should be the first step of the suggested solution that has been provided. requirements gathered using various methods or sources, as section I explains. Since we are aware of the client's lack of education and skill, we study the client's business system, staff members, and most importantly the end user who will be using the suggested system firsthand. We also meet with other staff members and company stakeholders to precisely determine what the proposed system's requirements are.

5.2. Filter Requirements

Filter the requirements once they have been gathered. Requirements are separated into functional and non-functional forms, sorted sequentially, and filtered to remove false, unclear, and unnecessary information.

5.3. Break Requirements

Once the requirements have been filtered, break them down into smaller, more manageable pieces. The software team found the requirement formulation and analysis to be quite simple to comprehend.

5.4. Prioritize Requirements

Sort the needs according to functionality at this stage. Requirements that are basic and essential

are given priority over non-essential requirements. Prioritize the development of the software's functionality. The early completion of priority needs results in increased client satisfaction.

4.5. Numbering the Requirements

The allocation of the numbering to slightly prioritize the criteria is the most crucial phase in this strategy. Typically, numerals like 1, 2, 3, 5, 6... n, are used for numbering. This will make it simple for us to compare the requirements to their quantity and begin working on them in the form of development.

4.6. Requirement Documentation

Ultimately, all of the criteria will be documented during this phase. Written requirements will be very helpful in understanding the system, tracking needs, all functionality, and how it is used, in the future (should the requirements change). The document will then be forwarded to the development team so that the software system may be developed. All processing needs to be recorded since, upon project completion, the client receives the recorded file, which explains each step, its use, and its potential future use.

6. RESULTS

To determine the pertinent outcomes, we apply the suggested solution to the Hotel Management System case study.

6.1. Requirement Gathering

Initially, we gather all 12 requirements for this case study, which are listed in table 1 below. (NADRA: National Database and Registration Authority).

Table 1: HMS Requirement Gathering

Hotel Management System	
Requirements	User login and password
	Book room on ID card
	Check authentication with NADRA
	Payment through cash/credit card
	Bank validation
	External reservation guest
	Check-out/ Check in
	Time scheduling
	Room cleaning staff
	Booking confirmation message/email
	Automated generated slip
	Food facility record

6.2. Filter Requirements

There are a total of 12 requirements, however, after sorting through them, we remove any unnecessary information and separate them into the functional and non-functional requirements for the hotel management system, as shown in table 2 below.

Table 2: HMS Functional/Non-Functional

Functional Requirements	Non-Functional Requirements
User login and password	External reservation guest
Book room on ID card	Booking confirmation message/email
Check authentication with NADRA	Room cleaning staff
Payment through cash/credit card	Time scheduling
Bank validation	Food facility record
Check-out/ Check in	-----
Automated generated slip	-----

6.3. Break Requirements

In this stage, we dissect each criterion into the smaller components listed in Table 2 above. But functional requirements are the main emphasis of our major.

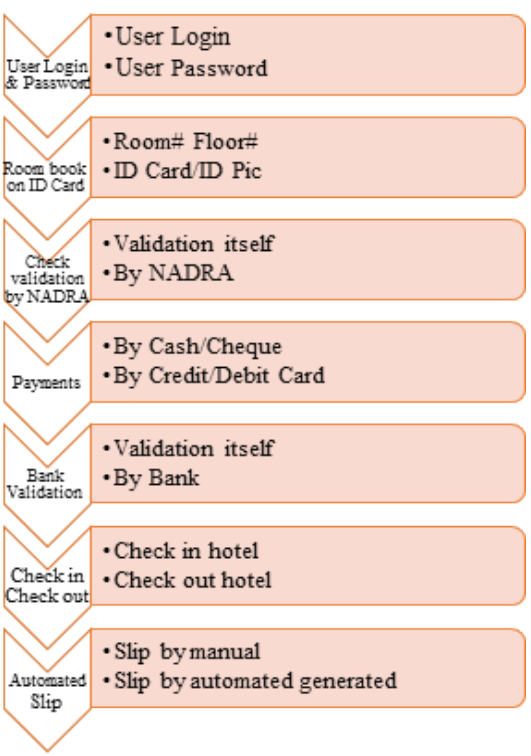

Figure 3: Break HMS Requirements

6.4. Prioritize Requirements

We will be ranking the seven functional needs of the HMS (Hotel Management System) shown in Table 2 above at this step of the suggested process.

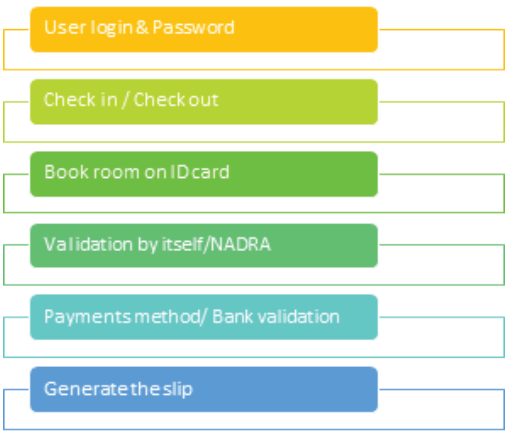

Figure 4: Prioritize Requirements

6.5. Numbering Requirements

The functional needs of the Hotel Management System, which are listed in Table 3 below, are being numbered in the virtually last stages of the proposed system.

Table 3: Numbering Requirements

Number Allocation	Requirements
1	User Login/Password
2	Check in/ Check out
3	Book the room
4	Validation by NADRA
5	Payments/Bank checking
6	Generate Slip

6.6. Requirement Documentation

At this stage, we will ultimately record every need listed in Table 3 and hand the project over to the Hotel Management software system development team.

7. DISCUSSION

This research paper assesses the process of code refactoring and its significance in improving the quality of software. Concerning code refactoring which is understood to be a constructive and effective approach to enhance program codes and performance, authors have not underestimated that it has variable effectiveness [22]. Therefore, the paper recommends that this approach be adjusted because the selection of the refactoring technique should relate to the nature and clarity of the requirements of the software. This way, it is seen that the majority of the issues can be handled in the early phases of the development life-cycle, provided that only the necessary and well-defined functional requirements are targeted, thus making the process of refactoring more effective [23].

The paper's section 4 considers the problems found in software companies today, especially the issue of code refactoring. Furthermore, in the foremost part of Section V, a solution is presented and it is highlighted the necessity of the refactoring strategies that are matched with the nature of the software project. This section takes a look into various methods used for verifying and fine-tuning the software requirements, along with keeping stakeholders involved in the procedure. As the last section of the paper, Section VI is to demonstrate results by using a case study. Thus, a hotel management system is chosen to apply the approach. The case study here demonstrates that the solution offered brings into being specific advancements in software quality, return on investment, and achievement of the project goals.

8. CONCLUSION

In this section, we conclude the paper that is staring at the effects of code refactoring on software quality. Everyone knows that who belongs to the software industry software quality is a big challenge for software survival and its credibility purpose [20]. Apply code refactoring in such a way that does not affect the software behavior and its output functionality. In another section, we will move to code refactoring to software requirements, because we know very well that every time code refactoring does not produce authentic results [24].

In the software requirement's structure, the whole development is based on it. It is an earlier phase of software development that uses the requirements-divided approach. Starting with gathering the requirements from actual sources, then filtering them, breaking them into relevant requirements, then allocating the numbers to break or filter requirements, and then all these requirements convert into documented form and send to the software development team. So, we can say that the requirements phase is the most valued in that we summarize the overall software project in the earlier phase in the requirements form. Due to the starting phase, it is not very expensive. If all

the requirements are valid and do not exist in any ambiguity, then the developed software is most probably error-free and based on client requirements.

9. FUTURE WORK

The future of code refactoring in software engineering will be mainly driven by automation, integrating these tasks with DevOps and investment into technical debt reduction, mature architectures modularization, security intensification, AI-based solution building, and educational ecosystem construction. These will be the key elements of more effective, fast-moving, and secure software development, at the end stage of improving the quality of software systems.

REFERENCES

[1] B. N. Van Vliet et al., "Direct and indirect methods used to study arterial blood pressure," *Journal of pharmacological and toxicological methods,* vol. 44, no. 2000, pp. 361–373, 2001.

[2] A. Hochstein et al., "Evaluation of Service-Oriented IT Management in Practice," *In Proceedings of ICSSSM'05. 2005 International Conference on Services Systems and Services Management,* Vol. 1, pp. 80-84, 2004.

[3] S. M. H. Dehaghani, and N. Hajrahimi, "Which Factors Affect Software Projects Maintenance Cost More?," *Acta Informatica Medica,* vol. 21, no. October 2012, pp. 63–66, 2013, doi: 10.5455/aim.2012.21, , 2013.

[4] M. Kessentini et al., "Many-Objective Software Remodularization using NSGA-III," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 24, no. 3, pp.1-45, 2015.

[5] E. F. Brown et al., "CRUSTAL HEATING AND QUIESCENT EMISSION FROM TRANSIENTLY ACCRETING NEUTRON STARS Edward F. Brown, Lars Bildsten, and Robert E. Rutledge 1," *The Astrophysical Journal,* vol. 504, no. 2, pp. 1996–1999, 1998.

[6] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," I*n 2012 28th IEEE international conference on software maintenance (ICSM),* pp. 306-315

[7] M. Abebe and C. Yoo, "Trends , Opportunities and Challenges of Software Refactoring : A Systematic Literature Review," *international Journal of software engineering and its Applications,* vol. 8, no. 6, pp. 299–318, 2014.

[8] S. Kaur and P. Singh, "The Journal of Systems and Software How does object-oriented code refactoring influence software quality ? Research landscape and challenges," *Journal of Systems and Softwar,* vol. 157, pp.110394, doi: 10.1016/j.jss.2019.110394, 2019.

[9] F. Coelho et al., "Refactoring-Aware Code Review : A Systematic Mapping Study," *In 2019 IEEE/ACM 3rd Int. Work. Refactoring*, pp. 63–66, doi: 10.1109/IWoR.2019.00019, 2019.

[10] A. Ouni et al., "Multi-Criteria Code Refactoring Using Search-Based Software Engineering : An Industrial Case Study," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 25, no. 3, pp.1-53, 2016.

[11] M. Paixão et al., "Behind the Intents : An In-depth Empirical Study on Software Refactoring in Modern Code Review," I*n Proceedings of the 17th International Conference on Mining Software Repositories,* pp. 125–136, 2020.

[12] B. Alshammari et al., "Security Assessment of Code Refactoring Rules," *In WIAR 2012; National Workshop on Information Assurance Research*, pp. 1-10, 2012.

[13] M. Alshayeb, "The Impact of Refactoring on Class and Architecture Stability," J*ournal of Research and Practice in Information Technology,* vol. 43, no. 4, pp.269-284, 2011.

[14] G. Szoke et al., "Designing and Developing Automated Refactoring Transformations : An Experience Report," *In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* Vol. 1, pp. 693-697, doi: 10.1109/SANER.2016.17, 2016.

[15] H. Mumtaz, et al., "An empirical study to improve software security through the application of code refactoring," *Inf. Softw. Technol.,* vol. 96, pp. 112–125, doi: 10.1016/j.infsof.2017.11.010, , 2018.

[16] E. Ammerlaan and A. Zaidman, "Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits," *In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* pp. 504–507, 2015.

[17] P. Techapalokul and E. Tilevich, "Code Quality Improvement for All: Automated Refactoring for Scratch," *2019 IEEE Symp. Vis. Lang. Human-Centric Comput.,* pp. 117–125, 2019.

[18] J. Cohen, "A COEFFICIENT OF AGREEMENT FOR NOMINAL SCALES," *Educational and psychological measurement,* vol. 20, no. 1, pp. 37–46, 2016.

[19] D. Damian et al., "An Industrial Case Study of Immediate Benefits of Requirements Engineering Process Improvement at the Australian Center for Unisys Software," *Empirical Software Engineering,* pp. 45–75, 2007.

[20] G. Lacerda et al., "Code smells and refactoring: A tertiary systematic review of challenges and observations," *J. Syst. Softw.,* vol. 167, pp.110610, doi: 10.1016/j.jss.2020.110610, 2020.

[21] A. Almogahed, Mahdin et al., "A refactoring categorization model for software quality improvement," *Plos one,* vol. 18, no. 11, pp. e0293742, 2023.

[22] B. Nyirongo et al., "A Survey of Deep Learning Based Software Refactoring," a*rXiv preprint arXiv:2404.19226,* 2024.

[23] E. A. AlOmar et al., "How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations," *In Proceedings of the 21st International Conference on Mining Software Repositories,* pp. 202-206, April 2024.

[24] A. C. Bibiano et al., "Composite refactoring: Representations, characteristics and effects on software projects," *Information and Software Technology,* 156, pp. 107134, 2023.

[25] N. Raeesinejad et al., *Refactoring. In The Ignite Project: A Journey in Scrum,* pp. 168-179, Singapore: Springer Nature Singapore, 2023.