Refactoring: Deep Dive and Current Trends

Braden Weaver
School of Computing
University of North Florida
Jacksonville, USA
n00914984@unf.edu

Sandeep Reddivari
School of Computing
University of North Florida
Jacksonville, USA
sandeep.reddivari@unf.edu

***Abstract*—Refactoring is an important activity in software engineering. Whether it takes place during the initial design of a system, or during maintenance, it is a powerful tool that can be used in order to keep code working and maintainable. Refactoring is used to eliminate Code Smells, in other words poorly designed code, while maintaining the functionality of the code. Keeping code understandable and well designed is critical to the success of any software project. In this paper, we present a literature review of recent advances and trends in refactoring and examine the practice of refactoring by looking at its key tenets.**

***Index Terms*—software, refactoring, maintenance, code smells**

I. Introduction

Refactoring is an important process during both the development and maintenance of a software system. As defined by Fowler and Martin, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [1]". In other words, refactoring is a process that can be used to ensure that a code base meets non-functional requirements such as maintainability, while preserving the functional aspects. Refactoring is important for a myriad of reasons. As a code base is grown and modified during its life-cycle due to changing requirements, the complexity of the system increases. This, in turn, contributes to lowering the overall quality of the code, as well as adding difficulty to maintenance [2]. This is known as software entropy.

Software entropy is directly combated by processes such as restructuring and refactoring, reducing the chaos and easing maintainability. This is especially important as the maintenance phase of software development life cycle (SDLC) accounts for a large amount of the total cost of a project, especially if the code needed drastic changes due to initial misunderstandings of requirements [3]. Studying the impacts of refactoring and how refactoring is used is important to further improving coding practices. AlOmar et al. [4] studied how developers documented refactoring activities in commit messages and looked for trends in commit messages relating to refactoring. Traini et al. [5] investigated the impact of refactoring on software performance in terms of execution time. Brito et al. [6] analyzed highly rated open source Java systems using the novel approach of refactoring graphs. Techapalokul and Tilevich [7] argue that manual refactoring, especially when attempted by novice programmers, is prone to introducing hard-to-trace bugs into a system. Mohan et al. [8] developed a tool called MultiRefactor with a multi-objective genetic algorithm in order to automate software refactoring and compared the multi and mono objective results in terms of coverage and quality.

II. Methodology

This paper is a survey of the recent research done in the field of software refactoring, as such a search was conducted using google scholar, IEEE Xplore, SpringerLink, and ACM Digital Library to locate papers using "Refactoring" as the search term. The search was limited to the past 6 years (2015-2021). The research in refactoring can be classified into two categories: automation tools and manual refactoring assistants.

Refactoring Tools: Batch Refactoring

Batch refactoring is the use of multiple refactorings together in a specific order with the use of tools. This can allow developers to improve quality over a broader area of the system they are working on, as shown in [8]. This is closely related to the automation of refactoring. Oliveira et al. [9] argued for the need of IDEs to allow developers the ability to customize batch refactoring. They posit that developers are dissatisfied with the customization options available with the current refactoring tools and often manually refactor as a result. It is their opinion that a niche is available for the addition of customization options to these IDEs in order to fix this problem. Their proposed approach is motivated by the lack of use of refactoring tools, despite them having been widely available for quite some time. Fernandes et al. [10] proposed several alternative methods for composing batch refactorings. Their position is that there was difficulty in composing batches that completely remove code smells. Their work was aimed at not only solving this issue, but also furthering the support of batch refactoring.

Refactoring Tools: Assistants

Alves et al. [11] developed a tool called RefDistiller in order to support developers who are using manual refactoring. They then tested RefDistiller with both seeded anomalies and a user study to confirm it's ability to detect problems and help developers with refactoring. The authors integrated Eclipse refactoring APIs into RefDistiller in order to provide a powerful tool to developers for determining if refactoring changes have altered the functionality of a program. Yoshida et al. [12] proposed an Eclipse plugin that monitors code modification in order to detect candidates for performing an Extract Method refactoring. The plugin then recommends this candidate to them for refactoring. The authors also performed a user study to compare their proposed plugin with Gemx, a code clone analysis environment. This user study found that their tool was faster than Gemx for making the refactoring changes. Alizadeh et al. [13] proposed an intelligent refactoring bot, called RefBot and it is integrated into the GitHub to continuously keeps track of the software repository. Antezana [14] proposed a tool called TOAD that can fix usability issues for refactoring tools. This tool was designed to guide users into selecting the correct code for Extract Method refactoring. TOAD was developed based on usability issues that they found with the refactoring tools in IntelliJ IDEA [14].

III. Discussion

While refactoring research is divided into themes, tools and practical research, these two are not entirely separate. Tool development continues to be a big focus of research even though tool usage by developers is not that high. Another aspect focused on is figuring out why developers are not using tools, and how to get them involved. Lastly, batch refactoring has been a popular theme, and is somewhat involved with the other two.

Manual Vs Automation: Developers are reluctant to make use of refactoring tools and that has become a frequently mentioned feature in refactoring research. Low tool adoption rates for a practice that is widespread is not very helpful for the software engineering community. Research has now shifted to learn more about why developers are not using refactoring tools and how to educate them to use these tools. Techapalokul and Tilevich [7] have shown that manual refactoring is prone to creating errors, especially when performed by a novice. Yet, the practice of manual refactoring continues, as many developers interleave refactoring work in with other development. This leads us directly to the next theme, as it is closely related to this argument.

Refactoring Tools: These are being developed to help software engineers deal with the ever increasing complexity of modern codebases in concurrence with a development of tools that help with manual refactoring. The incongruity here is quite interesting. The emergence of new tools that help suggest refactoring changes to developers and decrease usability problems with new tools promote both manual and fully automated refactoring. Increasing the intuitiveness of refactoring tools and processes will contribute greatly to software quality and longevity. The need for these tools is certainly increasing as the world moves towards an even greater reliance on software.

Batch Refactoring: It is an interesting parallel problem to the manual versus automation debate. It is a more complex and controlled version of automated refactoring giving software engineers a fine-grain control over the refactoring operation in their program. The limitation is that current IDEs do not fully support batch refactoring. This echoes the problems developers have had with refactoring tools since their inception, they lack the control, customization, and intuitiveness that is required. The combination of automated refactoring and fine-grain control over time and place multiple refactoring operations are performed will help immensely with the maintenance of complex software systems. Oliveira [9] showed, multi-objective refactoring has a much broader reach across a code base in about the same time as a single objective automated refactoring change. While the overall quality score does not go up as high, this process will still be able to help developers maintain large systems.

IV. Conclusion

Refactoring started as a way for object-oriented programmers to remove the code smells from their programs, and had evolved into an integral part of SDLC. The research on refactoring had been focused on improving the usability of refactoring tools in order to assist software engineers with software maintenance. In this paper, we presented the recent advances and trends in refactoring and examined the practice of refactoring by looking at its key tenets.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[3] A. M. Davis, *Software requirements: objects, functions, and states*. Prentice-Hall, Inc., 1993.

[4] E. AlOmar, M. W. Mkaouer, and A. Ouni, “Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 51–58, IEEE, 2019.

[5] L. Traini, “How software refactoring impacts execution time,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–23, 2021.

[6] A. Brito, A. Hora, and M. T. Valente, “Refactoring graphs: Assessing refactoring over time,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 367–377, IEEE, 2020.

[7] P. Techapalokul and E. Tilevich, “Position: Manual refactoring (by novice programmers) considered harmful,” in *2019 IEEE Blocks and Beyond Workshop (B&B)*, pp. 79–80, IEEE, 2019.

[8] M. Mohan, D. Greer, and P. McMullan, “Maximizing refactoring coverage in an automated maintenance approach using multi-objective optimization,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 31–38, IEEE, 2019.

[9] D. Oliveira, A. C. Bibiano, and A. Garcia, “On the customization of batch refactoring,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 13–16, IEEE, 2019.

[10] E. Fernandes, A. Uchôa, A. C. Bibiano, and A. Garcia, “On the alternatives for composing batch refactoring,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 9–12, IEEE, 2019.

[11] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, “Refactoring inspection support for manual refactoring edits,” *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 365–383, 2017.

[12] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, “Proactive clone recommendation system for extract method refactoring,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 67–70, IEEE, 2019.

[13] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, “Refbot: intelligent software refactoring bot,” in *2019 34th IEEE/ACM international conference on automated software engineering (ASE)*, pp. 823–834, IEEE, 2019.

[14] A. S. Antezana, “Toad: A tool for recommending auto-refactoring alternatives,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 174–176, IEEE, 2019.