AI Assisted/Automated code refactoring

How the development of AI may impact the future of code refactoring

Loris Tomassetti
Linz, Austria
loris.tomassetti@outlook.com

Alexander Weißenböck
Linz, Austria
alewei934@gmail.com

***Abstract*—This Paper aims to shed light onto developments in AI Assisted/Automated Code refactoring, how it can help the indtustry and which models work most efficiently to tackle different challenges code refactoring brings. This will be done by going over various literature describing first the challenges at hand and afterwards discussing several possible solutions that have been tested to gain a greater understanding and to generate an informed outlook into further developments of this technology.**

***Index Terms*—machine learning algorithms, software code refactoring, deep neural network**

I. Introduction

Refactoring, as defined by Fowler [3], is "the process of changing a software system in such a way that does not alter the external behavior of the code yet improves its internal structure". More and more empirical studies have since established a positive correlation between refactoring operations and code quality metrics. All this evidence hints at refactoring being a high-priority concern for software engineers. [2].
However, deciding when and how to refactor can prove to be a challenge for developers. Refactoring in an early stage may be cost too much for what youre getting out of it, and refactoring too late may cause the refactor to be an even bigger time comittment. [5]
Tools have been in the hands of many developers to make this process more streamlined for years now. Analytics tools to sniff out bugs or give hints on how to improve code quality such as PMD, ESLint, and Sonarqube can be integrated in different stages of a developers' workflow, e.g. inside IDEs, during code review or as an overall quality report. [2]
Taking a closer look at these tools, however, reveals that they commonly have a lot of false positives, making developers lose their confidence in them. Often, the detection strategies are based on hard threshholds of just a handfull of metrics, such as lines of code in a file (e.g. PMD's famous "problematic" classification occuring once a method reaches 100 lines per default). These simplistic ways of detection simply aren't able to capture the full complexity of modern software systems.
Manually analyzing hundreds of metrics and figuring out which ones are the cause of technical dept is very hard and almost impossible for tool-developers, which is where machine learning-based solutions come into play.
We will take a closer look at how exactly different ML-Models go about this task in section IV

II. Refactoring

A. Uses of Refactoring

Fowler states refactoring describes changing the internal structure of code without changing its behaviour. [3] This can lead to improvements in code in various aspects. By reducing the complexity of code while keeping its behavior intact it becomes more human-readable and therefore also more maintainable since it is easier to expand on it in the future. [4] Another great advantage of refactoring is the reduction of technical debt. This term refers to a metaphorical debt when subpar solutions have been chosen to accelerate the production of software. While the chosen solutions might work they often do not take into account further developments and through this often cause problems in the future. [6] And of course, the overall code quality can be improved. The main difficulty in code refactoring lies in finding places to optimize, otherwise called "bad smells". These bad smells can and therefore refactoring can happen on different levels within the code, like Class level refactoring describing for example the extraction of a class into one or more sub-classes, while a variable level refactoring could be something like executing an action inline instead of creating an unnecessary variable. [2] This makes it very difficult to locate these bad smells and as a consequence time consuming and expensive.

III. Approaches for Automation

A. Large Language Models

1) GPT Model:

2) Github co-pilot:

3) Fauxpilot Client:

B. Dedicated Models

1) DNNFFz:

IV. Benefits of AI-Powered Refactoring

Logically, automated refactoring comes with all the benefits dicussed in the prior chapter, namely more readable, higher quality code that is more maintainable, but more importantly it makes all this possible in a shorter amount of time. [**?**] Automating the refactoring process is not a new idea at all and numerous tools have been developed and modern IDE's already support some refactorings. However those are often small scale and need input on what to refactor. [8] It could

also be verified that autmated refactorings bring a statistically significant time improvement to nearly every type of refactoring. [7] This of course means less cost in extension. The same study also found that currently automated refactoring is more often used for smaller chnages, since there the refactoring is less error prone. Using AI the possible usecases could be significantly expanded and lead to greater time savings.

V. Challenges and Limitations

A. Over-reliance on Automation

Especially when using applications like Chat GPT users often rely too much on the model understanding their problem and give insufficient and/or vague requests which have a lot of room for interpretation. This can then, unsurprisingly, lead to undesired results. [**?**]

B. Potential for Unintended Consequences

A big reason software developers often choose to not use atomated refactoring is the unpredictability of the result. "[...] If I cannot guess, I don't use the refactoring. I consider it not worth the trouble. [...]" [8] Since refactoring can span multiple files it can be hard for a developer to check all the changes after an automatic refactoring has occured. Especially generative AI solutions which do not only handle smell detection but also the generation of solutions have often found to provide mixed results, which can lead to the behavior of the code being changed. This also leads to reduced trust to those tools by developers. [**?**]

C. Performance Concerns

While big strides in the technology are being made, it is still a present issue that some automatic refactorings do produce errors especially when misused [1] Especially when using AI based tools it is not always given that the behaviour of the code is indeed the same as before, making the refactoring itself faulty. Baqais and Alshayeb could find quite a lack in checking for bahavior in a multitude of research papers, leaving out a crucial step. [**?**]

D. Trust and knowledge gaps

Some developers stated, when using an AI refactoring tool, it would probably be useful to know how it works in the background to potentially get better results. [**?**] This would then require more focus on learning those Models. Also in another study Developers did not use certain automated refactorings of IDE's because they did not know of their existence. [7] In both of those cases better education about already existing tools alone could help improve performance.

VI. Future Directions and Research Opportunites

Due to these discussed limitations there are two main takeaways for problems that must be tackled for AI powered refactoring to be truly industry changing. Firstly there has to be more work put into the systems to avoid bahavioural changes of the refactored software. Secondly it will be very important to improve the communication between tools and develorpers as well as improve education about existing tool so that they can be used to their full potential.

A. Personalized Code Refactoring Suggestions

VII. Discussion

VIII. Methodology

IX. Conclusion and Outlook

References

[1] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[2] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450, 2020.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.

[4] Amandeep Kaur and Manpreet Kaur. Analysis of code refactoring impact on software quality. In *MATEC Web of Conferences*, volume 57, page 02012. EDP Sciences, 2016.

[5] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[6] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[7] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 552–576. Springer, 2013.

[8] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, 2012.