Iterative Refactoring of Real-World Open-Source Programs with Large Language Models

Jinsu Choi, Gabin An, and Shin Yoo(✉)

KAIST, Daejeon, Republic of Korea
{jinsuchoi,agb94,shin.yoo}@kaist.ac.kr

Abstract. Code refactoring is a critical task for improving software quality, but it is traditionally a manual, time-consuming process. This paper demonstrates an approach to automate project-level code refactoring using Large Language Models (LLMs). The key idea is to iteratively identify methods with high cyclomatic complexity, and then use LLMs to generate refactored implementations that reduce complexity. Our evaluation using 17 open-source projects shows that the proposed automated refactoring can reduce average cyclomatic complexity by up to 10.4% within 20 iterations. These results suggest that automated project-level code refactoring is feasible using LLMs with tailored prompts.

Keywords: Code Refactoring · Large Language Model · Cyclomatic Complexity

1 Introduction

Code refactoring is the process of modifying a software system's internal structure without changing its external behavior [1]. Traditionally, code refactoring tasks are performed manually by developers based on their experience and industry best practices [2], which can make large-scale refactoring a time-consuming process that takes weeks to months to complete.

Recently, there has been growing interest in applying Large Language Models (LLMs) to various Software Engineering tasks, such as code generation, test generation, and automated debugging [3]. Prior work [4,5] has noted that LLMs can be effectively combined with traditional search-based software engineering techniques because this combination allows LLMs to provide more powerful and tailored code mutations, while the generate-and-validate approach helps prevent LLMs from generating unreliable or hallucinated outputs. Code refactoring is particularly well-suited for leveraging LLMs, as the assurance criteria can be fully automated using a regression test oracle [4]. However, existing LLM-based refactoring work has relied only on existing tests to check the correctness of refactoring patches, without the use of such regression oracles [6]. Additionally, this study has evaluated performance using only introductory programs, rather than real-world software projects.

G. Jahangirova and F. Khomh (Eds.): SSBSE 2024, LNCS 14767, pp. 49–55, 2024.
https://doi.org/10.1007/978-3-031-64573-0_4

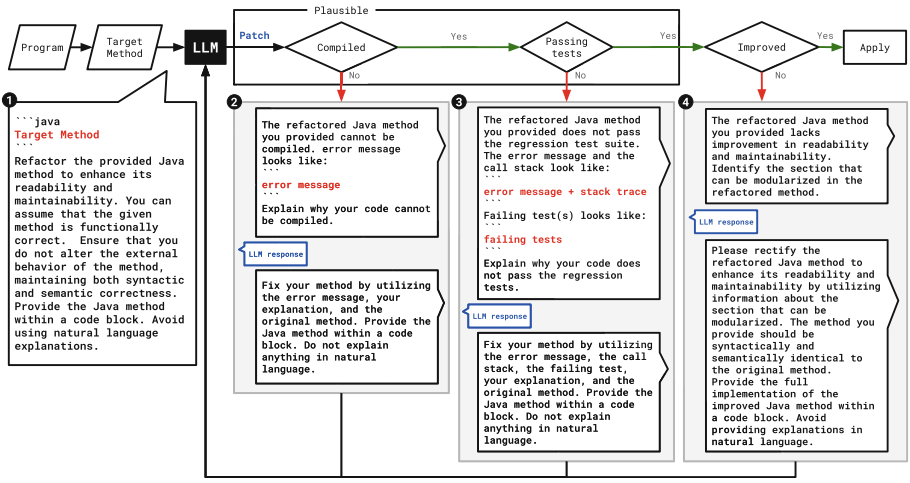

Fig. 1. Overview

This paper demonstrates the use of LLMs to perform iterative project-level code refactoring with an objective of reducing the Cyclomatic Complexity (CC) [7], the number of linearly independent paths within the source code. The approach first identifies the methods with the highest CC and then performs refactoring on them. Once an improved version of the method implementation is found, it is applied to the codebase, and the approach subsequently attempts to refactor the next high-complexity method. To prevent the existing functionality from breaking during the refactoring, we use automated generated regression tests to filter out incorrect patches and best assure that the refactored code maintains the original behavior [4]. To evaluate our approach, we use 17 open-source projects from the Defects4J benchmark [8]. The results demonstrate that our method can reduce the average CC of the programs by up to 10.4% within 20 iterations. We observe that LLMs sometimes significantly reduce the CC by splitting a complex method into multiple simple methods. Overall, these findings suggest the viability of LLM-based automated refactoring for large-scale software projects.

2 Methodology

Our methodology for iteratively reducing the code complexity of a target program is illustrated in Fig. 1. The process begins by identifying the method with the highest CC in the project. Subsequently, the LLM is requested to refactor this selected method (Sect. 2.1). The refactored code passes through a series of filters. To ensure that the refactoring does not break the existing functionality of the original code, the plausibility of the refactored code is verified using two types of regression tests: developer-written tests and automatically generated tests (Sect. 2.2). If the LLM has indeed improved the code quality, the

refactored code is applied to the project (Sect. 2.3). This iterative process is repeated, progressively enhancing the overall code quality of the project.

2.1 Extract and Refactoring Target Method

We use the Lizard library[1] to measure the CC of all methods within the source files of the project. After identifying the method with the highest CC, we retrieve the source code of that particular method. The retrieved method implementation is then embedded into the prompt (❶ in Fig. 1) which is provided as input to the LLM. The prompt instructs the LLM to improve the readability and maintainability of the method implementation, while ensuring that the refactored method remains semantically equivalent to the original one.

2.2 Checking Plausibility

After the LLM generates the refactoring patch, we verify the plausibility of the refactored method implementation. This involves checking whether the program is still compilable and whether it successfully passes both the developer-written test suite and the automatically generated regression tests. Note that the regression tests were created with respect to the original program before any refactoring was applied. If the refactored method is implausible, we request the LLM to rectify the issues. In case of a compilation failure, the prompt supplies the compilation error message to the LLM (❷ in Fig. 1). If the compilation succeeded but the tests did not pass, the prompt provides both a stack trace and details of the failing tests, in addition to the error message (❸ in Fig. 1). After presenting this error information, the prompt requires the LLM to analyze the root cause of the issue. Subsequently, the prompt requests the LLM to fix the method, i.e., regenerate the refactoring patch. If the fixed method remains implausible, meaning it still fails the checks, we exclude that method from the improvement targets and proceed to the next iteration without applying the refactored method in the project.

2.3 Assessing Improvement

If a plausible patch is found, we measure the CC of the refactored method and compare it to the CC of the original method. If the LLM has split the original method into multiple methods, we compare the CC of the method with the highest complexity. If the CC is reduced, the refactored method is then applied to the target project, and we move forward with the next iteration. This iterative process allows the project to be continuously improved. If the CC of the refactored method does not decrease, i.e., there is no improvement in complexity, we request the LLM to identify a part of the code that can be further modularized (❹ in Fig. 1). Subsequently, the LLM is tasked with enhancing the quality of the code based on its own analysis and response. If the modified method is

[1] https://github.com/terryyin/lizard.

Table 1. Refactoring results: the percentage reduction in the average CC, the percentage increase in the number of functions, and the percentage reduction in the average number of lines of code. The maximum values across the five attempts are presented.

Project	Avg. CC Reduction	# Func Increase	Avg. nLoC Reduction	Project	Avg. CC Reduction	# Func Increase	Avg. nLoC Reduction
Chart	0.27%	0.10%	0.74%	Cli	4.94%	5.70%	4.67%
Closure	0.35%	0.24%	0.21%	Codec	2.37%	4.41%	2.94%
Collections	0.06%	0.08%	0.06%	Compress	1.15%	0.32%	0.96%
Csv	10.40%	17.86%	10.60%	Gson	2.95%	3.32%	2.46%
JacksonCore	0.38%	0.54%	0.41%	JacksonDatabind	0.10%	0.11%	0.11%
JacksonXml	2.31%	3.10%	2.79%	Jsoup	0.88%	1.44%	1.03%
JxPath	0.79%	0.58%	0.99%	Lang	0.80%	0.67%	1.22%
Math	0.01%	0.02%	0.01%	Mockito	0.78%	0.97%	0.53%
Time	0.84%	0.42%	0.35%				

still not plausible or fails to improve the complexity, we exclude that method from future iterations and proceed to the next iteration without applying the refactored method in the project.

3 Experimental Setup

We evaluate our LLM-based refactoring pipeline using the 17 real-world Java projects from Defects4J v2.0.0 [8]. As multiple snapshots of each project are available in the benchmark, we utilize the latest version for every project. For our experiment, we used the `gpt-3.5-turbo-0125`. Due to the stochastic nature of the LLM querying process, we run our pipeline five times for each project, with each execution comprising 20 iterations of refactoring. To generate regression tests for the method in focus at each iteration, we make use of the *gen_tests* script of Defects4J. In particular, EvoSuite [9] is utilized to generate tests for Java classes that contain the target method, with an allocated time budget of 180 s. If the generated regression tests lead to failures when run against the program, we eliminate those test cases and initiate the test generation process again, allowing for a maximum of five attempts. If failures persist beyond five attempts, it is assumed that the target method (or class) contains elements of non-determinism, leading us to halt further attempts to improve the method.

4 Results

Refactoring Results: We examine the extent to which the average CC of the entire project decreases as the iteration progresses. Figure 2 illustrates the change in average CC across projects over successive iterations. In every project, there is at least one instance where the average CC decreases. On average, our method reduces the average CC of the project by 1.2%. Table 1 displays more

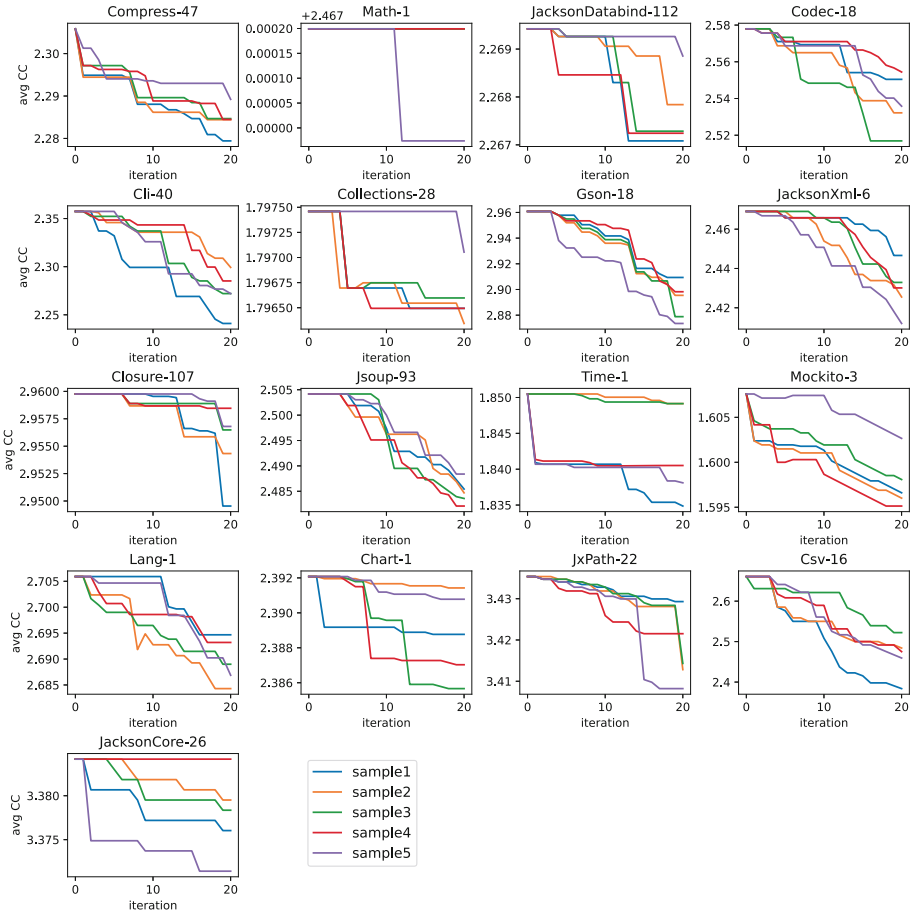

Fig. 2. Change in average CC over 20 iterations

detailed refactoring results after the 20 iterations. The highest reduction rate of average CC is 10.4% for Csv-16. During the refactoring process, there are instances where the target method is split into multiple methods, resulting in the number of functions in the project increasing by 1.5% on average across all refactoring attempts. Notably, in Csv-16, where the average CC decreased the most, the number of functions increased by 17.9%. As the method is separated, the average length of the method is shortened: the average number of Lines of Code without comments (nLoC) is reduced by 1.3% on average. The examples of LLM-generated refactoring patches are available online at https://figshare.com/s/6e7d9f69a96974c110fb.

Detailed Statistics of the LLM-Generated Code: During our experiment, the LLM is requested to refactor the method a total of 1,700 times across all projects. We found that 45.5% of the refactoring attempts initially generate

non-plausible methods. Among them, 14.0% of the non-plausibility can only be discovered through automated regression tests. The plausibility fixing process leads to 33.7% of the initially non-plausible methods becoming plausible. After this fixing process, 69.8% of all attempts produce plausible methods. However, among these, 81.7% show no improvement. Further modularization process leads to improvement in 30.9% of these initially non-improved methods.

5 Conclusion and Future Work

In this study, we explore the potential of using LLMs to enhance the quality of program source code, specifically aiming to lower CC. We discovered that integrating LLMs with our iterative search methodology and tailored prompts successfully decreases the project's average CC and enhances its modularity, all while preserving its original functionality, which is confirmed through both developer-written and automated regression testing. However, manual verification is still necessary to ensure that the code remains semantically identical. Furthermore, our current pipeline has been limited to the sequential refactoring of individual methods. Moving forward, we aim to develop a more structural refactoring approach using LLMs, such as the detection and modularization of recurring code patterns across different methods as well as operations that involve multiple methods. We also plan to assess the effectiveness of LLM-based refactoring on other non-functional quality metrics, such as execution time and memory usage. Finally, we will investigate whether LLMs can guide more qualitative refactoring goals, instead of simply following quantitative metrics [10].

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
2. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012)
3. Fan, A., et al.: Large language models for software engineering: survey and open problems. arXiv preprint arXiv:2310.03533 (2023)
4. Alshahwan, N., Harman, M., Harper, I., Marginean, A., Sengupta, S., Wang, E.: Assured LLM-based software engineering (2024)
5. Kang, S., Yoo, S.: Towards objective-tailored genetic improvement through large language models. In: 2023 IEEE/ACM International Workshop on Genetic Improvement (GI), pp. 19–20, IEEE (2023)
6. Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M., Watanobe, Y.: Refactoring programs using large language models with few-shot examples. arXiv preprint arXiv:2311.11690 (2023)
7. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **SE–2**(4), 308–320 (1976)
8. Just, R., Jalali, D., Ernst, M.D.: Defects4J: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, New York, NY, USA, pp. 437-440, Association for Computing Machinery (2014)

9. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, New York, NY, USA, pp. 416–419. ACM (2011)
10. Simons, C., Singer, J., White, D.R.: Search-based refactoring: metrics are not enough. In: Barros, M., Labiche, Y. (eds.) SSBSE 2015. LNCS, vol. 9275, pp. 47–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22183-0_4