Duplicate Code Refactoring via Just-In-Time

Akshar Awari Ivan Jacobs Cody Wilson

DSCI-644 Team 4

{aba7569, ij6019, ctw3883}@rit.edu

1. Introduction

The process of copying code fragments from foreign code base or one's own code base into other code base levels (such as the service layer, data layer, or distinct classes) is known as source code duplication and is a widespread practice. Due to the normalization of this technique by social programming platforms through code search and reuse in other projects, there is a significant quantity of source code duplication (for example, 70% of GitHub is made up of copies of previously published source code) [1]. According to the estimates in the industry, source codes contain 5-20% of duplicated snippets [12]. The expenses of changes made after delivery are projected to account for 40% to 70% of all expenditures incurred over the course of a system's lifespan, indicating the severe financial impact of maintenance [12]. The developer and scientific communities acknowledge that this phenomena poses a variety of difficulties for code maintenance and metrics reporting for code models also gets skewed [2]. Hence researchers are looking to understand cross and in-project code reuse [3]–[6] in order to provide means and tools [1] to tackle this challenge. Elimination of the duplicate code and replacement of duplicate code with new methods or functions is critical to improving the code base in a number of ways. Therefore, in order to propose ways and tools [1] to address this difficulty, researchers are trying to comprehend cross and in-project code reuse. The code base may be improved in a variety of ways by getting rid of duplicate code and replacing it with new methods or functions.

All development environments use refactoring, a controlled way to modify an existing code base without changing its behavior, to enhance design and performance. Refactoring duplicate code, often known as the "Extract Method," specifically entails refactoring duplicate code fragments into new methods or services and replacing them with calls to the new source code. Many refactoring goals and techniques, including Move Method possibilities discovery, code smell reduction, technical department identification, antipattern detection, and impact [7]–[11] have been well researched and offer candidates that may be used to address code duplication.

Current studies on utilizing the Extract Method of refactoring are limited to after-the-fact analyses. They are to be performed after code has been written and the developer is intentionally seeking to refactor code. The Just-In-Time methodology combines the act of writing new code and refactoring to eliminate duplicate code into one task. While the developer is writing code, a prebuilt model analyzes and detects when duplicate code is being written, and determines if it is worth combining with previous code to create a method to share the workload. This can be incredibly beneficial because the developer is actively engaged in the current problem, meaning they generally are at a higher level of understanding of the current code and functionality. When refactoring at a later date, that intimate knowledge of the problem at hand is diminished,

[1] https://github.com/skyhover/Deckard

meaning more mental energy and uncertainty is exerted when determining if refactoring duplicate code is logical and productive.

[1] C. V. Lopes *et al.*, “DéjàVu: a map of code duplicates on GitHub,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, p. 84:1-84:28, Oct. 2017, doi: 10.1145/3133908.
[2] “The adverse effects of code duplication in machine learning models of code | Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.” https://dl.acm.org/doi/10.1145/3359591.3359735 (accessed Aug. 27, 2022).
[3] “Cross-project code clones in GitHub | Semantic Scholar.” https://www.semanticscholar.org/paper/Cross-project-code-clones-in-GitHub-Gharehyazie-Ray/5812937e05870571ff3cd1bb6c31029372305dac (accessed Aug. 27, 2022).
[4] “Some from Here, Some from There: Cross-Project Code Reuse in GitHub.” https://ieeexplore.ieee.org/document/7962379/ (accessed Aug. 27, 2022).
[5] B. HU, Y. WU, X. PENG, J. SUN, N. ZHAN, and J. WU, “Assessing code clone harmfulness: Indicators, factors, and counter measures,” *2021 28th IEEE Int. Conf. Softw. Anal. Evol. Reengineering Virtual March 9-12 Proc.*, pp. 225–236, Mar. 2021, doi: 10.1109/SANER50967.2021.00029.
[6] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.
[7] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, “Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems,” in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10. doi: 10.1109/ICSM.2010.5609564.
[8] “When and why your code starts to smell bad | Proceedings of the 37th International Conference on Software Engineering - Volume 1.” https://dl.acm.org/doi/10.5555/2818754.2818805 (accessed Aug. 27, 2022).
[9] N. Tsantalis and A. Chatzigeorgiou, “Identification of Move Method Refactoring Opportunities,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May 2009, doi: 10.1109/TSE.2009.1.
[10] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns,” *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, Apr. 2011, doi: 10.1016/j.jss.2010.11.921.
[11] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, Jun. 2012, doi: 10.1007/s10664-011-9171-y**.**
[12] Mayrand, Leblanc and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," 1996 Proceedings of International Conference on Software Maintenance, 1996, pp. 244-253, doi: 10.1109/ICSM.1996.565012.