EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs

Dorin Pomian*
abhiram.bellur@colorado.edu
University of Colorado Boulder
Colorado, USA

Abhiram Bellur*
dorin.pomian@colorado.edu
University of Colorado Boulder
Colorado, USA

Malinda Dilhara
malinda.malwala@colorado.edu
University of Colorado Boulder
Colorado, USA

Zarina Kurbatova
zarina.kurbatova@jetbrains.com
JetBrains Research
Belgrade, Serbia

Egor Bogomolov
egor.bogomolov@jetbrains.com
JetBrains Research
Amsterdam, the Netherlands

Andrey Sokolov
andrey.sokolov@jetbrains.com
JetBrains Research
Amsterdam, the Netherlands

Timofey Bryksin
timofey.bryksin@jetbrains.com
JetBrains Research
Limassol, Cyprus

Danny Dig
danny.Dig@colorado.edu
University of Colorado Boulder
Colorado, USA

ABSTRACT

Excessively long methods, loaded with multiple responsibilities, are challenging to understand, debug, reuse, and maintain. The solution lies in the widely recognized *Extract Method* refactoring. While the application of this refactoring is supported in modern IDEs, recommending which code fragments to extract has been the topic of many research tools. However, they often struggle to replicate real-world developer practices, resulting in recommendations that do not align with what a human developer would do in real life. To address this issue, we introduce *EM-Assist*, an IntelliJ IDEA plugin that uses LLMs to generate refactoring suggestions and subsequently validates, enhances, rank, and apply them. In our evaluation of 1,752 real-world refactorings that actually took place in open-source projects, EM-Assist's recall rate was 53.4% among its top-5 recommendations, compared to 39.4% for the previous best-in-class tools. Moreover, we conducted a usability survey with 18 industrial developers, and 94.4% gave a positive rating.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**;
• **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Refactoring, LLMs, Code smells, Long Methods, Java, Kotlin

ACM Reference Format:
Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. *EM-Assist*: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663803

*These authors contributed equally to this work.

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663803

1 INTRODUCTION

Long methods encapsulate multiple responsibilities and are challenging to comprehend, debug, reuse, and evolve [2, 15, 28]. They often lead to code that is not only difficult to understand but also error-prone, thereby becoming a significant source of technical debt in software projects [15]. To alleviate this, developers frequently use the *Extract Method* refactoring, which divides methods into smaller, more manageable units. This refactoring consistently ranks among the top five most commonly performed in practice [19, 20, 29].

The *Extract Method* refactoring process comprises two phases: (i) Selecting statements for extraction from the original method, involving a meticulous examination of code segments that encapsulate specific responsibilities or logic, with the goal of isolating them into a distinct method, (ii) Applying the refactoring, involving moving the selected statements into a brand new method, passing necessary variables as parameters, and invoking the new method from the original context. While the application part has been a staple feature of all modern IDEs, they leave it up to developers to choose which statements to extract. The research community has developed diverse techniques to suggest statements for extraction. Some research tools use static analysis in conjunction with software quality metrics [6, 18, 27, 28, 31], such as statement cohesion [28], while others employ machine learning-based classifiers [7, 30]

While the existing tools adhere to software quality metrics based on software engineering principles, they often generate suggestions that do not align with real-world *Extract Method* instances. We attribute this to the fact that refactoring requires both technical rigor and subjective discernment. In other words, developers rely on their understanding of software engineering principles and their subjective interpretation of code context when deciding what makes a good method. While existing techniques excel in the former, they often fall short in the latter. This gap may explain developers' reluctance to use automated refactoring support [26].

To bridge the gap and generate *Extract Method* suggestions that developers are likely to accept, we utilize Large Language Models

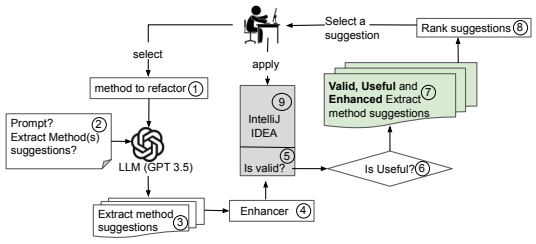

Figure 1: The workflow of generating refactoring suggestions

(LLMs). They are trained on extensive code repositories containing millions of methods authored by actual developers. Thus, they are more likely to mimic human behavior and replicate how developers create methods, making them likely to propose refactorings that developers would embrace [10, 22]. Our formative study, conducted on an extended corpus size of 1,752 *Extract Method* real refactorings from open-source systems, demonstrated that LLMs are highly effective in providing expert suggestions, generating 27 suggestions per method. This highlights the substantial potential of integrating LLMs into the domain of code refactoring. Nevertheless, LLM output cannot be directly adopted. Our findings indicate that 76.3% of LLM suggestions are *hallucinations* of two kinds. First, 57.4% of the LLM suggestions are invalid (e.g., syntactically incorrect), potentially causing compilation errors. Second, 18.9% of the suggestions are not useful (e.g., suggest to extract the entire method body).

To advance the field of refactoring, we introduce *EM-Assist*, the *first automated refactoring tool to use LLMs. EM-Assist* is an IntelliJ IDEA plugin. It employs LLMs to generate refactoring suggestions, then filters those that are invalid and non-useful, and further enhances valid suggestions using program slicing techniques. It then ranks them to offer high-quality options to developers. Moreover, *EM-Assist* bridges the gap between suggesting and applying refactorings by encapsulating selected suggestions into refactoring commands and executing them within the IDE. This process leverages the IDE to ensure the safe execution of refactorings, providing a seamless transition from suggestion to execution.

In section 3.1, we first quantitatively evaluate the *EM-Assist* in terms of its effectiveness compared to baseline tools. Then in section 3.2 we qualitatively evaluate *EM-Assist*'s usability. In evaluations, we observed that EM-Assist really shines over previous approaches when replicating real-world refactorings in contemporary codebases. When replicating 1,752 actual *Extract Method* refactorings that took place in open-source projects, EM-Assist's recall rate was 53.4%, compared to 39.4% for the previous best-in-class tool that relies solely on static analysis (JExtract). For our qualitative evaluation (section 3.2), we surveyed 18 industrial developers, confirming the tool's usability at an approval rate of 94.4%.

2 *EM-ASSIST*

2.1 Example Usage of *EM-Assist*

Figure 2 shows an example of using *EM-Assist* to suggest and execute refactorings on the method `writeJvmClass()` within the project *JetBrains/intellij-community*. The user triggers *EM-Assist* by choosing the "Extract method suggestions" option from the intention actions (yellow light bulb) [**?**] for the method (① in Figure 2). In a popup window, *EM-Assist* shows three distinct suggestions for extracting code fragments, accompanied by a suggested method name (② in Figure 2). In this example, the plugin suggests three extract methods: `writeInterfaces()`, `writeFields()`, and `writeMethods()`, and it indicates the size of the code fragment for each suggestion. To make it convenient for the user to assess each suggestion, *EM-Assist* helps the user preview the signature of the proposed new method (including method parameters) directly within the popup window. Additionally, for each suggestion, *EM-Assist* highlights the code fragment slated for extraction. For example, the blue-colored statements in Figure 2 (line 85-90 of ①) represent the proposed statements for extraction within the `writeMethods()` suggestion. Likewise, the plugin affords flexibility to users, allowing them to choose the most appropriate extract method suggestion and seamlessly apply it to the `writeJvmClass()` method. To do so, the user clicks the "Extract" button conveniently located within the popup window. In this example, the user chose to apply the `writeMethods` suggestion to the code, and the resulting code is shown as ③ in Figure 2.

2.2 Workflow

In this section, we present the internal workflow that *EM-Assist* uses to automatically suggest and perform *Extract Method*. As shown in Figure 1, the developer invokes the plugin by clicking the "Extract Method" option in the intention actions for a particular method they intend to apply the extract method refactoring to (① in Figure 1). Then *EM-Assist* creates a LLM few-shot learning prompt (② in Figure 1) that contains the method declaration, and it invokes the LLM programmatically to generate refactoring suggestions (③ in Figure 1). To tame the non-determinism of LLMs, *EM-Assist* repeats the same prompt request, using optimal values that we determined empirically (i.e., between 5-10 iterations). Then *EM-Assist* enhances the LLM suggestions by adjusting the code fragments, using program slicing (④ in Figure 1). However, not all these enhanced suggestions are safe to apply, as some may result in non-compilable and erroneous code. In fact, our analysis conducted on a dataset of 1,752 methods, reveals that 76.3% of the suggestions generated by the LLM are invalid, potentially resulting in non-compilable code. To filter invalid suggestions, *EM-Assist* employs the IntelliJ IDEA APIs for determining whether a code fragment meets the refactoring preconditions for extract method (⑤ in Figure 1).

Some valid suggestions are not practical for real-world implementation (e.g., those that contain the whole method body). Therefore, *EM-Assist* filters unuseful suggestions (⑥ in Figure 1). In the end, *EM-Assist* generates numerous suggestions per long method (⑦ in Figure 1), which could potentially overwhelm the user if all are presented at once. To address this, we employ a ranking mechanism (⑧ in Figure 1) that relies on the frequency of LLM suggestions. The ranking mechanism is centered on the idea that if a particular code fragment is repeatedly suggested for extraction by the LLM during subsequent triggers, it must represent an important functional concern that is suitable to be in a separate method. Then, *EM-Assist* presents the ranked suggestions (top-n, default is top-3) to developers through our user interface, offering previews of extracted method signatures and associated code fragments. After the user selects their preferred suggestion, *EM-Assist* executes the refactoring correctly (⑨ in Figure 1).

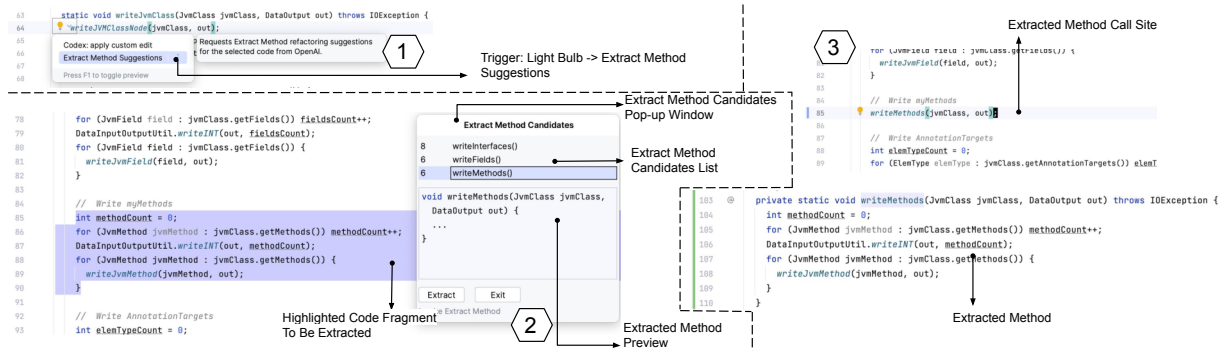

Figure 2: Workflow for using *EM-Assist* within IntelliJ IDEA: 1) the user triggers the plugin to generate suggestions, 2) the plugin displays three refactoring options in a popup window, 3) the user selects one of the options and inspects the final code.

2.3 Implementation

EM-Assist is implemented as an IntelliJ IDEA plugin and works as an intention action. IntelliJ Platform has a suite of APIs for Abstract Syntax Tree (AST) manipulation and refactoring. *EM-Assist* leverages IntelliJ's AST manipulation through the Program Structure Interface (PSI), which is the layer in the IntelliJ Platform responsible for code, file, and project model. *Extract Method* refactoring is applied through IntelliJ's refactoring framework. We employ few-shot learning [5, 23] to form the LLM prompt. *EM-Assist* is configurable to use various LLMs, by default we use OpenAI's GPT-3.5-turbo. The communication with LLM is asynchronous, thus, the IntelliJ code editor is not frozen and multiple requests are sent to the LLM in parallel. Not all suggestions provided by the LLM are valid and ready to be extracted, therefore, we use PSI elements to adjust the scope of these *Extract Method* candidates. We utilize IntelliJ's Extract Method framework to further filter out the candidates which cannot be extracted. We enhanced the Extract Method framework to support custom method names suggested by the LLM.

Extensions: *EM-Assist* currently supports Java and Kotlin, but its implementation offers an extensible framework for supporting other languages as well. For example, to support *Extract Method* in Python code, extensions would be needed in steps ①, ③, ⑤, and ⑦ (Figure 1), whereas ②, ④, ⑥, ⑧, ⑨ (Figure 1), could be reused. To support other IDEs that provide good support for ExtractMethod refactoring, steps ⑤ and ⑨ (Figure 1) would need change, whereas all remaining steps could be reused. Moreover, our preliminary experiments with using LLMs to suggest other kinds of refactorings show that LLMs are prolific, but we need to employ a similar approach: use the LLM for developer-aligned suggestions, but carry out the refactoring plan with the safety of the IDE.

Limitations: Currently, using *EM-Assist* requires sending the user's host method code to OpenAI. As a result, the response time can vary. Our experiments involving 25,235 invocations of the tool revealed that, on average, *EM-Assist* takes 2 seconds to suggest refactorings. This includes the time taken to invoke the LLM and the additional processing time required by the tool. According to Nielsen's usability guidelines [21], this response time is acceptable for human-operated software tools, ensuring that the user's flow of thought remains uninterrupted.

3 EVALUATION

We conducted a comprehensive evaluation of *EM-Assist*, using a harness of two complementary approaches. (i) Quantitative (section 3.1): we compared the effectiveness of EM-Assist with two state-of-the-art extract method recommendation tools by replicating a large dataset of 1,752 actual extract methods performed by open-source developers. Unlike previous studies that used a small, synthetic corpus of artificially-created ExtractMethods, we are replicating real-world refactorings from famous, contemporary open-source projects. We believe this is more indicative of a tool's capabilities in dealing with the complexities of real-world refactorings, and the large scale of experiments guards against over-fitting a tool for a small corpus. (ii) Qualitative (section 3.2): we evaluated the usability of *EM-Assist*'s approach when guiding industrial developers to follow our novel workflow for performing ExtractMethod refactorings. Unlike previous studies that have employed large number of students, we believe our study employing 18 full–time, experienced professional developers is more indicative of the quality of the suggested refactorings.

3.1 Effectiveness of *EM-Assist*

To determine *EM-Assist*'s effectiveness in recommending refactorings that align with developer preferences, we compare it to previous state-of-the-art approaches that rely solely on static analysis and software quality metrics. In our technical report [22], we compare *EM-Assist* to six other tools that suggest *Extract Method* and we use a synthetic corpus used by other researchers. We found that JExtract [25] outperforms the previous tools, thus we use JExtract as the strongest competitor to *EM-Assist*.

First, we construct an oracle. We mined the complete version history of projects using RefactoringMiner [29], a state-of-the-art tool for refactoring detection. Thus we identified 1,752 cases when the open-source developers performed ExtractMethod refactorings. We then replicate all these refactorings by running the tools on the original host methods (the version before refactoring was applied) and compare the suggested refactorings against the actually-performed refactorings in the oracle. The oracle exhibited a diverse range of method involved in refactoring, with min/max/mean/median values of 3/1494/30/5 LOC for the host method, and 2/95/6/3 LOC for extract methods. Given the size of host methods, there can be many ways to group code fragments for extraction.

We executed the two tools on each host method in the oracle. We then compared the refactorings performed by these tools and cross-referenced them with the oracle to ensure that the extracted code lines matched those in the oracle. Following best practices established in prior research [6, 7, 14, 25, 28, 30] we evaluate top-5 suggestions generated by the tools, calculating Recall@5 at tolerance level 3%. We selected Recall as the performance metric as this is the metric used by all previous tools. Given the size of host methods, even at 3% tolerance, it means in many cases we must have an exact match between one of the top-5 suggestions and the extracted code fragment in the oracle. We compute the recall by dividing the total number of refactoring suggestions that matched the oracle to the total number of extract methods in the oracle.

EM-Assist has a recall rate of 53.4%, significantly surpassing its counterparts whose best recall was 39.4%. Acknowledging the probabilistic nature of *EM-Assist*'s underlying LLM, we conducted further statistical analysis for robust validation. We repeated each prediction from *EM-Assist* 50 times and performed a one-sample t-test against *JExtract*, the second-best performing tool. The test decisively rejected the null hypothesis asserting that *JExtract* had a better recall rate than EM-Assist, with a p-value of $< 10^{-5}$.

3.2 Usability of *EM-Assist*

We complement the previous quantitative evaluation with a survey to evaluate developers' perception of using the novel workflow of *EM-Assist* in comparison to the current workflow in the IDEs. We intentionally designed the survey to allow participants to view a brief, feature-focused demo of the plugin, rather than requiring them to install and execute it. Previous research [8, 16] demonstrated that this approach is useful for evaluating the fundamental concepts and ideas behind a tool, as high-fidelity prototypes might overwhelm survey participants with excessive implementation details.

We surveyed 18 qualified professional software developers. Among these participants, 50% have more than a decade of experience in software development, 33% have between one and three years of experience, while others have between three and seven years of experience. We used Likert-type questions [4] to evaluate *EM-Assist*'s features and adhered to established best practices [3].

The respondents expressed highly positive attitudes towards the overall usability and user interfaces of *EM-Assist*. 94.4% of the respondents found the additional workflow stages, including triggering the plugin, generating LLM suggestions, and selecting one to apply, to be convenient and easy to handle. *EM-Assist* uses a popup window to present the suggestions, and over 77.7% of respondents agreed that it is somewhat helpful or greater. Additionally, 94.4% of respondents liked the suggested method names. While users could get name suggestions from an LLM at a later time, we conveniently offer meaningful names at the time of performing extract method.

In the open-ended questions, the respondents suggested valuable enhancements to *EM-Assist* in terms of usability, configurability, and mode of execution, which we will use in a future release.

Overall, we received positive praises from the survey respondents about *EM-Assist*: *"It looks super cool so far! :fire:"*. Another said: *"Thank you for interesting suggestions! Hope to see this in production in the future."* And another: *"...these suggestions made me look at this code with new eyes once more, and I will try to refactor it."*

4 RELATED WORK

Researchers developed tools to suggest code fragments to extract. Many primarily rely on static analysis-based rules like program slicing [1, 17, 18], the separation of concerns [24], or the single responsibility principle [6]. While these tools excel at adhering to software quality principles, they face a significant limitation. They are not able to adapt suggestions to developers' subjective intentions, possibly because of their lack of access to real-world data illustrating actual developer refactoring practices.

As an alternative to rule-based techniques, machine learning-based classifiers [7, 30] have been proposed for refactoring suggestions. Despite having access to relevant data, they grapple with practical challenges such as data scarcity, protracted training periods, and resource-intensive training procedures. Furthermore, these ML models are often specific to particular tasks, necessitating periodic retraining to maintain their effectiveness [9, 11–13]. In contrast to these existing tools, *EM-Assist* leverages LLMs as a promising solution. LLMs are general-purpose models pre-trained on extensive source code and text, enabling them to generate refactoring suggestions aligned with developer intentions using knowledge from both code and textual documents. *EM-Assist* takes this a step further by removing hallucinations, validating and enhancing the LLM output, and providing developers with tailored recommendations..

While obtaining the training set for every kind of refactoring is a formidable challenge for previous approaches, it is not for LLM-based tools. While previous approaches require continuous/custom model training or custom analysis, given the broad knowledge-base of LLMs, our solution is easier to use, maintain, and deploy in IDEs.

5 CONCLUSIONS

We present *EM-Assist*, the first automated refactoring tool that employs LLMs, implemented as an IntelliJ IDEA plugin. It bridges the gap between suggesting and applying *Extract Method* refactoring, and shrinks the gap between refactoring suggestions and developer practices. *EM-Assist* generates refactoring suggestions using LLMs, then validates and enhances them with static analysis in the IDE. To avoid overwhelming developers, *EM-Assist* ranks and presents only the highest-quality suggestions, and then correctly executes the user-selected refactoring suggestion with the IDE. We comprehensively evaluate *EM-Assist* and it outperforms state-of-the-art. Additionally, a developer survey confirmed the high usability of the plugin. *EM-Assist* uses a novel way to check LLM results and make them useful for refactoring tasks. Being the first tool to employ LLMs for *Extract Method* refactoring, *EM-Assist* demonstrates the potential of LLMs as effective refactoring assistants. We are now expanding *EM-Assist* to support many more kinds of refactorings.

ACKNOWLEDGEMENTS

We thank the ML Methods in Software Engineering Lab at JetBrains Research, and the FSE-2024 reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grants CNS-1941898, CNS-2213763, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.

REFERENCES

[1] Aharon Abadi, R Ettinger, and YA Feldman. 2009. Fine slicing for advanced method extraction. *3rd workshop on refactoring tools.*

[2] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. 1993. Software complexity and maintenance costs. *Commun. ACM.*

[3] Carol M Barnum. 2010. Usability testing essentials. *Elsevier.*

[4] Harry N Boone Jr and Deborah A Boone. 2012. Analyzing likert data. *The Journal of extension.*

[5] Tom et. al Brown. 2020. Language Models are Few-Shot Learners. *Curran Associates, Inc.*

[6] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *TSE.*

[7] Di Cui, Qiangqiang Wang, Siqi Wang, Jianlei Chi, Jianan Li, Lu Wang, and Qingshan Li. 2023. REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph. In *ICPC.*

[8] Scott Davidoff, Min Kyung Lee, Anind K Dey, and John Zimmerman. 2007. Rapidly exploring application design through speed dating. In *UbiComp.* Springer.

[9] Malinda Dilhara. 2021. Discovering repetitive code changes in ML systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 3 pages.

[10] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *FSE* (2024). arXiv:2402.07138

[11] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).*

[12] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.*, Article 55 (July 2021), 42 pages.

[13] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering repetitive code changes in python ML systems. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 13 pages.

[14] Sara Fernandes, Ademar Aguiar, and André Restivo. 2022. A Live Environment to Improve the Refactoring Experience.

[15] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[16] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid moves between code and graphical work in computational notebooks.

[17] Arun Lakhotia and Jean-Christophe Deprez. 1998. Restructuring programs by tucking statements into functions. Elsevier.

[18] Katsuhisa Maruyama. 2001. Automated Method-Extraction Refactoring by Using Block-Based Slicing.

[19] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *TSE.*

[20] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. *ECOOP.*

[21] Jakob Nielsen. 1994. *Usability engineering.* Morgan Kaufmann.

[22] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring. arXiv:2401.15298 [cs.SE]

[23] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog.*

[24] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. *ICPC.*

[25] Danilo Silva, Ricardo Terra, and Marco Túlio Valente. 2015. Jextract: An eclipse plug-in for recommending automated extract method refactorings.

[26] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. *FSE.*

[27] Omkarendra Tiwari and Rushikesh Joshi. 2022. Identifying Extract Method Refactorings. *ISEC.*

[28] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods.

[29] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *TSE.*

[30] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. GEMS: An Extract Method Refactoring Recommender. *ISSRE.*

[31] Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying Fragments to Be Extracted from Long Methods. *APSEC.*

Received 2024-01-29; accepted 2024-04-15