HECS: A Hypergraph Learning-Based System for Detecting Extract Class Refactoring Opportunities

Luqiao Wang
Xidian University
Xi'an, China
wangluqiao@stu.xidian.edu.cn

Qiangqiang Wang
Xidian University
Xi'an, China
qiangqiangwang@stu.xidian.edu.cn

Jiaqi Wang
Xidian University
Xi'an, China
jiaqi@stu.xidian.edu.cn

Yutong Zhao
University of Central Missouri
Warrensburg, USA
yutongzhao@ucmo.edu

Minjie Wei
Xidian University
Xi'an, China
weiminjie@stu.xidian.edu.cn

Zhou Quan
Xidian University
Xi'an, China
zquan@stu.xidian.edu.cn

Di Cui
Xidian University
Xi'an, China
cuidi@xidian.edu.cn

Qingshan Li
Xidian University
Xi'an, China
qshli@mail.xidian.edu.cn

Abstract

HECS is an advanced tool designed for Extract Class refactoring by leveraging hypergraph learning to model complex dependencies within large classes. Unlike traditional tools that rely on direct one-to-one dependency graphs, HECS uses intra-class dependency hypergraphs to capture one-to-many relationships. This allows HECS to provide more accurate and relevant refactoring suggestions. The tool constructs hypergraphs for each target class, attributes nodes using a pre-trained code model, and trains an enhanced hypergraph neural network. Coupled with a large language model, HECS delivers practical refactoring suggestions. In evaluations on large-scale and real-world datasets, HECS achieved a 38.5% increase in precision, 9.7% in recall, and 44.4% in f1-measure compared to JDeodorant, SSECS, and LLMRefactor. These improvements make HECS a valuable tool for developers, offering practical insights and enhancing existing refactoring techniques.

CCS Concepts

• **Software and its engineering** → **Software Maintenance**.

Keywords

Extract Class Refactoring, Hypergraph Neural Network

ACM Reference Format:
Luqiao Wang, Qiangqiang Wang, Jiaqi Wang, Yutong Zhao, Minjie Wei, Zhou Quan, Di Cui, and Qingshan Li. 2024. HECS: A Hypergraph Learning-Based System for Detecting Extract Class Refactoring Opportunities. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685307

*Qingshan Li and Di Cui are the corresponding authors.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685307

1 Introduction

Large classes that encompass multiple responsibilities within a single unit pose significant challenges in terms of comprehension and maintenance [14]. To mitigate this issue [23, 30], developers employ the Extract Class refactoring technique. This method decomposes monolithic classes into smaller, more manageable units, generating new classes that can be reused. The Extract Class refactoring process consists of two main phases: (i) identifying appropriate fields or methods to extract and (ii) implementing the refactoring mechanics to reorganize the class's internal structure without altering its external behavior [33].

Several automatic Extract Class refactoring tools have been proposed [6, 8, 9, 16], these tools build an intra-class dependency graph with nodes representing fields or methods and edges for field-access or method-call dependencies. They use heuristic rules based on this graph to suggest refactoring opportunities. However, traditional intra-class dependency graphs focus on one-to-one relationships and struggle with complex one-to-many dependencies. As a result, their refactoring suggestions may not always align with developers' preferences, highlighting the need for more sophisticated analysis to better capture and suggest meaningful refactoring opportunities.

To enhance the accuracy and comprehensiveness of refactoring suggestions, we introduce the **H**ypergraph learning-based **E**xtract **C**lass refactoring detection **S**ystem, HECS[1][2]. HECS uses hypergraphs to represent dependencies, allowing edges to bind multiple nodes and reflect complex interactions among fields and methods. This enables HECS to identify cohesive groups for precise refactoring suggestions. By mining patterns from historical samples, HECS automates the detection and suggestion of Extract Class

[1]https://github.com/cznzb/HECS
[2]https://doi.org/10.5281/zenodo.12662219

refactoring opportunities, providing highly relevant and accurate recommendations.

HECS transforms intra-class dependency graphs into hypergraphs and attributes nodes in these hypergraphs using pre-trained code models. It then employs an enhanced hypergraph neural network for training. This trained network, combined with a large language model (LLM), powers a system that identifies fields and methods to extract into new classes. Tested on large-scale and real-world datasets [18, 34], HECS outperforms three SOTA tools: JDeodorant [9], SSECS [8], and LLMRefactor [29] in effectiveness and usefulness. HECS offers an intuitive interface for easy integration into existing workflows. Additionally, we have published a research paper describing HECS and evaluation in detail [11].

2 HECS Design

The workflow of HECS is presented in Fig. 1, which consists of four phases: (1) hypergraph construction, (2) node attribute generation, (3) refactoring opportunity suggestion, and (4) LLM-based pre-condition verification. This comprehensive workflow enables HECS to identify and recommend potential areas for code refactoring effectively.

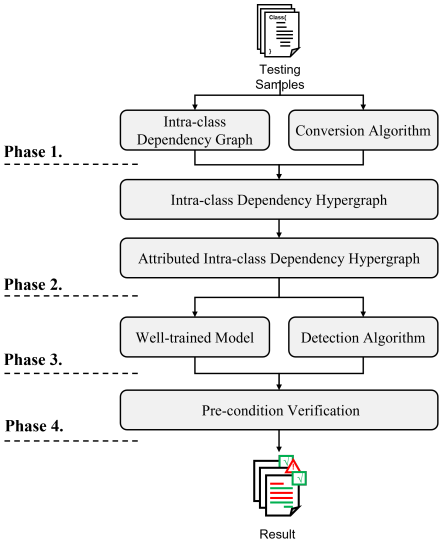


Figure 1. Workflow of HECS.

Hypergraph Construction. HECS begins by transforming a given program into a hypergraph, extracting the intra-class dependency graph using *ENRE* [17], a state-of-the-art static analysis tool that supports over 11 types of code dependencies. *ENRE* provides flexible interfaces for implementing custom analyzers efficiently. Inspired by the concept of design rule spaces [35], which model code dependency graphs as multiple overlapping spaces, we developed a heuristic algorithm to discover these design spaces and convert them into hypergraphs. Alg. 1 takes an intra-class dependency graph G_c with vertex set V_c and edge set E_c as input and outputs an intra-class dependency hypergraph $\mathcal{G}_c$.

Node Attribute Generation. In the hypergraph $\mathcal{G}_c$, each vertex represents a code snippet, and its attributes are generated using pre-trained code models. Including CodeBERT, GraphCodeBERT, CodeGPT, CodeT5, CoTexT, and PLBART, are known for their robust code representation capabilities. Each line of code within a snippet is individually embedded by these models, resulting in a set of embeddings: $\{ebd(c_i)|i = 1, 2, \cdots, k\}$. To create a comprehensive node attribute, we apply mean-pooling across these embeddings, producing a node attribute $x_i \in \mathbb{R}^{768}$, where $d = 768$ as heuristically determined. The final node attribute space $\mathbf{X}_c = \{x_i|i = 1, 2, \cdots, n\}$ forms the input for our hypergraph learning process. Our tool thus integrates these advanced code models to automatically generate meaningful representations for code analysis, enhancing the detection and understanding of intra-class dependencies.

Algorithm 1 Intra-class Dependency Hypergraph Construction

Input: $G_c = (V_c, E_c)$ - Intra-class dependency graph
Output: $\mathcal{G}_c$ - Intra-class dependency hypergraph

```
E_c^* ⟵ TransC(E_c)                ▷ The transitive closure of E_c
for each v_i in V_c do
    he_i ⟵ v_i
    for each v_j in V_c do
        if (v_j, v_i) ∈ E_c^* then
            he_i.append(v_j)
        end if
    end for
    G_c.add(he_i)                  ▷ Construct the hyperedge for each vertex
end for
return G_c
```

Refactoring Opportunity Suggestion. HECS inputs a target class to extract its attributed intra-class dependency hypergraph, which is then processed by a trained model to generate candidate fields and methods. Inspired by hierarchical clustering, our tool uses a heuristic algorithm to identify refactoring candidates. Given a target class tc, the attributed hypergraph $\mathcal{AG}tc$, the set of fields/methods $\mathcal{FM}tc$, and the number of extracted classes r, the algorithm proceeds as Alg. 2. Each field/method is initialized as an individual cluster. The matrix $\mathbf{P}$ is used to store the likelihood of merging between clusters, initially set to $\emptyset$. All cluster pairs are generated and the likelihood score is calculated with the well-trained model. Clusters are iteratively merged until the desired number of extracted classes is reached. The cluster pair $(clus_i, clus_j)$ with the highest score $score_h$ in matrix $\mathbf{P}$ is identified. If $score_h$ is less than 0.5, no further classes can be extracted. The cluster pair $(clus_i, clus_j)$ is merged into a new cluster $clus_{new}$. The clusters and matrix $\mathbf{P}$ are updated accordingly. The resulting clusters are collected as the final refactoring candidates.

LLM-based Pre-Condition Verification. HECS includes a critical phase called LLM-based Pre-Condition Verification. This phase ensures that refactoring does not introduce bugs or compromise semantic consistency. Based on initial definition [24] and refactoring practice [4, 28], HECS formally verifies pre-conditions for each Extract Class refactoring candidate to ensure behavior preservation and functional usefulness. The pre-conditions are:

- **P1:** Abstract methods should not be extracted.
- **P2:** Constructors should not be extracted.
- **P3:** Delegate methods should not be extracted.
- **P4:** Methods that override an abstract or concrete method of superclass should not be extracted.
- **P5:** Fields accessed by other classes should not be extracted.

Algorithm 2 Refactoring Opportunity Suggestion Algorithm

Input: $\mathcal{AG}_{tc} = \{\mathcal{V}_{tc}, \mathcal{E}_{tc}, \mathbf{X}_{tc}\}$ - Attributed intra-class dependency hypergraph of target class: tc
Input: $\mathcal{FM}_{tc} = \{fm_i | i = 1, 2, \cdots, n\}$ - Fields/methods of tc
Input: r - Number of extracted classes
Output: *ECSet* - Set of extracted classes

$clusters \leftarrow \{\{fm_i\} | i = 1, 2, \cdots, n\}$ ▷ Initialize clusters
$\mathbf{P} \leftarrow \emptyset$ ▷ Initialize proximity adjacent matrix
for all $pair$ in `AllPair`$(clusters)$ **do**
  $\mathbf{P}[pair[0]][pair[1]] \leftarrow$ `Model`$(\mathcal{AG}_{tc}, pair)$
end for
while $|clusters| > r$ **do**
  $score_h, (clus_i, clus_j) \leftarrow$ `MaxValue`$(\mathbf{P}, clusters)$
  if $score_h < 0.5$ **then**
    return $\emptyset$ ▷ No classes can be extracted
  end if
  $clus_{new} \leftarrow$ `Merge`$(clus_i, clus_j)$
  $clusters \leftarrow clusters - \{clus_i\} - \{clus_j\}$
  for all $clus_k$ in $clusters$ **do**
    $\mathbf{P}[clus_k][clus_{new}] \leftarrow$ `Model`$(\mathcal{AG}_{tc}, (clus_{new}, clus_k))$
  end for
  $clusters \leftarrow clusters + \{clus_{new}\}$
end while
$ECSet \leftarrow clusters$
return *ECSet*

P6: Methods that have any super method invocations should not be extracted.
P7: Methods that are synchronized or have a synchronized block should not be extracted.
P8: The extracted class should contain more than one field or method.

Unlike previous approaches that use heuristic rules/static code analysis [4, 7, 13, 19, 20]. HECS employs a LLM to adapt to various cases through prompt engineering. This approach consists of two steps. In Step 1, the LLM uses few-shot learning to extract pre-condition relevant fields/methods. In Step 2, the LLM checks whether these entities violate the pre-conditions using a verification prompt. HECS utilizes GPT-3.5 [26] and three instances for each pre-condition in few-shot learning. The prompts and instances used are available for reference. For each refactoring candidate, HECS filters out invalid candidates through Step 2 prompt engineering with the LLM to provide the final refactoring suggestions.

3 Deployment and Usage of HECS

HECS[3] utilizes a two-layer client-server architecture: the client layer facilitates user interaction, while the server layer handles refactoring opportunity detection.

The Client Layer. The Client Layer of HECS is a Visual Studio Code extension that interacts with the server layer. After installation, users can access the client layer via a launch button, as shown in Fig. 2. It includes *Detection View* and *Refactoring View*.

- *Detection View*: In Fig. 2.(a), users select a source file and click the "detection" button (Button 1) to send an HTTPS request to the server layer. The results, summarized in a table listing field or method names.
- *Refactoring View*: In Fig. 2.(b), clicking the "preview" button (Button 2) opens a new window showing code differences between the target and extracted classes, color-coded with green for added and red for deleted code using diff2html. Clicking the "refactoring" button (Button 3) automatically applies the detected refactoring opportunities.

The Server Layer. HECS's server layer, deployed on Azure Kubernetes Service [32], listens for HTTPS requests and returns detection results. Implemented in Python and PyTorch [27], it comprises three modules: code analysis, representation generation, and refactoring detection. Usage steps are provided below:

Download:

- Obtain the .vsix file from the provided website.

Installation:

- Open Visual Studio Code and navigate to the Extensions view (Ctrl+Shift+X).
- Click the menu button in the upper right corner of the Extensions view.
- Select "Install from VSIX" and use the file browser to locate and open the downloaded .vsix file.

Run Analysis:

- Click the HECS icon in the bottom-right corner to access the plugin interface.
- Select the Java source file and the desired refactoring operation.
- Click "detection" to generate code refactoring suggestions, "preview" to review the proposed changes, and "refactoring" to apply the modifications.

4 Evaluation

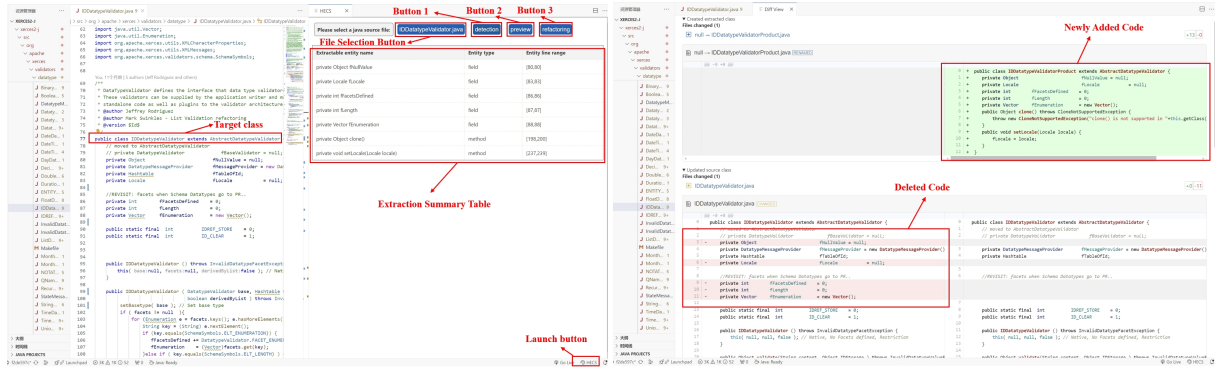
In this section, we evaluate HECS and compare it with state-of-the-art Extract Class refactoring tools, including LLMRefactor [29], SSECS [8], and JDeodorant [9]. Experiments were conducted on a 2.4GHz Intel Xeon-4210R server with 10 logical cores, 128GB of memory, and 4 NVIDIA RTX 4090 GPUs. We used the default hyper-parameters from previous work [10] and implemented all neural networks with the Python library.

Evaluation Metrics. We use Precision, Recall, and F1-Measure to evaluate HECS, following previous refactoring-related work [36]. For multiple class extraction, we construct a weighted bipartite graph with extracted classes of ground-truth and detection results. The edge weights are the number of overlapping fields or methods. The Hungarian algorithm [22] is used to find the maximum weighted matching, and evaluation metrics are calculated based on this matching.

Experiment Result. We conduct a systematic comparison of HECS on 12 combinations of 6 pre-trained code models and 2 hypergraph neural networks. We further regard the most effective combination HGNN$^+$+CodeBERT as HECS.

Table 1 presents the average accuracy of each refactoring tool on each instance of *AccEval*. We highlight the greatest precision, recall, and f1-measure score with red color. As presented in Table 1, we observed that 1) LLMRefactor demonstrates a higher recall score, while SSECS achieves a greater precision score. Furthermore, JDeodorant outperforms both LLMRefactor and SSECS in terms of

[3] https://www.youtube.com/watch?v=-Jtss1eKo5U



(a) A snapshot of detection view

(b) A snapshot of refactoring view

Figure 2. The graphical user interface of our prototype implementation as a VS Code extension.

the f1-measure score. 2) HECS demonstrates an increase of 38.5% in precision, 9.7% in recall, and 44.4% in f1-measure compared to the best results of LLMRefactor, JDeodorant, and SSECS. The improvement of HECS can be attributed to effective capture of refactoring characteristics with hypergraph neural network.

Table 1: Performance of refactoring tools on *AccEval* [34].

Refactoring Tool	Precision	Recall	F1-Measure
LLMRefactor	19.8%	**74.6%**	31.3%
JDeodorant	24.1%	71.2%	**36.0%**
SSECS	**38.3%**	16.1%	22.7%
HECS-β	**76.8%**	**84.3%**	**80.4%**

To evaluate the usefulness of HECS, we conducted a user study involving 50 industrial engineers. The participants were recruited using a snowball sampling method, initiated through company contacts known to the authors. The results present that 42% of participants reported 4 to 5 years of experience in software development. Only 10% of participants never considered Extract Class refactoring during development and 30% of participants reported that Extract Class refactoring can be conducted without the aid of tools. After reviewing all the suggestion results on *ECHumanEval* [34] for four refactoring tools, 64% of participants recommend HECS compared to other refactoring tools.

5 Tool Availability

Download and Access. The latest version of HECS, including usage instructions and guidelines, is available on GitHub [3].
Demonstration Video. A demo video of HECS is available [2].
Archived Version. An archived version of HECS at the time of submission is available on Zenodo [1].

6 Related Work

Extract Class Refactoring Approaches. Opdyke et al. [25] initially designed an Extract Class refactoring algorithm using aggregations and reusable components. Simon et al. [31] introduced the Jaccard distance between methods to identify such opportunities. Gabriele et al. [9] enhanced this with a weighted graph and subgraph segmentation based on Jaccard distance. Akash et al. [6] improved detection by incorporating semantic dependence between methods through cosine similarity of topic distributions. Zhao et al. [37] introduced the Butterfly Space method, leveraging both static and dynamic method-level dependencies. Jeba et al. [16] used context metrics and hierarchical clustering for better accuracy. Akash [5] applied Variational Graph Auto-Encoders (VGAE) and clustering to suggest refactorings.

Machine learning-based Refactoring Approaches. Xu et al. [36] used machine learning to identify Extract Method opportunities, encoding metrics like complexity and cohesion as features. Mikolov et al. [21] used word2vec and Convolutional Neural Networks (CNN) to detect code smells. Hadj-Kacem et al.[15] utilized Variational Auto Encoders (VAE) to extract features from abstract syntax trees for logistic regression classifiers. Cui et al. [12] employed graph embedding techniques to learn features and train classifiers for guiding Move Method refactoring.

7 Conclusion

In this paper, we introduced HECS, a novel tool for suggesting Extract Class refactorings by leveraging historical data. HECS begins by extracting intra-class dependency graphs, converting them into intra-class dependency hypergraphs, and employing an enhanced hypergraph neural network alongside with a large language model (LLM) to provide refactoring suggestions. Our findings demonstrate that HECS achieves a 38.5% increase in precision, a 9.7% increase in recall, and a 44.4% increase in F1-measure compared to leading tools such as JDeodorant, SSECS, and LLMRefactor. Additionally, 64% of participants found HECS more practical for real-world applications. Overall, HECS represents a significant advancement in Extract Class refactoring by integrating hypergraph neural networks with large language models, thus contributing a substantial improvement to software engineering tools.

Acknowledgements

This work was partially funded by National Natural Science Foundation of China (U21B2015,62202357), Proof of Concept Foundation of Xidian University Hangzhou Institute of Technology (XJ2023230039), Natural Science Foundation of Jiangsu Province (BK202302028).

References

[1] 2024. *HECS Archived Version.* https://doi.org/10.5281/zenodo.12662219
[2] 2024. *HECS Demo Video.* https://www.youtube.com/watch?v=-Jtss1eKo5U
[3] 2024. *HECS Source Code.* https://github.com/cnznb/HECS/
[4] Marios Fokaefs A, Nikolaos Tsantalis A, Eleni Stroulia A, and Alexander Chatzigeorgiou B. 2012. Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software* 85, 10 (2012), 2241–2260.
[5] Pritom Saha Akash and Kevin Chen-Chuan Chang. 2022. Exploring Variational Graph Auto-Encoders for Extract Class Refactoring Recommendation. arXiv:2203.08787
[6] P. S. Akash, A. Sadiq, and A. Kabir. 2019. An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity. In *International Conference on Evaluation of Novel Software Approaches to Software Engineering.*
[7] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation.* IEEE, 104–113.
[8] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. 2014. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.
[9] G. Bavota, A. D. Lucia, and R. Oliveto. 2011. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems & Software* 84, 3 (2011), 397–414.
[10] 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 *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC).* IEEE, 191–202.
[11] Di Cui, Qiangqiang Wang, Yutong Zhao, Jiaqi Wang, Minjie Wei, Jingzhao Hu, Luqiao Wang, and Qingshan Li. 2024. One-to-One or One-to-Many? Suggesting Extract Class Refactoring Opportunities with Intra-class Dependency Hypergraph Neural Network. In *Proceedings of the 33nd ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA).* ACM.
[12] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 281–292.
[13] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. 2020. On the relationship between refactoring actions and bugs: a differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 556–567.
[14] Martin Fowler. 1997. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.
[15] Mouna Hadj-Kacem and Nadia Bouassida. 2019. Deep representation learning for code smells detection using variational auto-encoder. In *2019 International Joint Conference on Neural Networks (IJCNN).* IEEE, 1–8.
[16] T. Jeba, T. Mahmud, P. S. Akash, and N. Nahar. 2020. God Class Refactoring Recommendation and Extraction Using Context based Grouping. *International Journal of Information Technology and Computer Science* 5 (2020).
[17] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. 2019. ENRE: a tool framework for extensible eNtity relation extraction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings.* IEEE Press, 67–70.
[18] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *2009 Ninth International Conference on Quality Software.* 305–314. https://doi.org/10.1109/QSIC.2009.47
[19] Tom Mens, Gabriele Taentzer, and Olga Runge. 2007. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling* 6, 3 (2007).
[20] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. 2005. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 4 (2005), 247–276.
[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[22] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.
[23] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18.
[24] William F Opdyke. 1992. *Refactoring object-oriented frameworks.* University of Illinois at Urbana-Champaign.
[25] W. F. Opdyke and R. E. Johnson. 1992. Refactoring Object-Oriented Frameworks. *University of Illinois at Urbana-Champaign* (1992).
[26] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Curran Associates Inc., Red Hook, NY, USA.
[28] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. 2010. Correct refactoring of concurrent Java code. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24.* Springer, 225–249.
[29] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples. *arXiv preprint arXiv:2311.11690* (2023).
[30] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering.* 858–870.
[31] F. Simon, F. Steinbruckner, and C. Lewerentz. 2001. Metrics based refactoring. In *European Conference on Software Maintenance & Reengineering.*
[32] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining.* 2727–2735.
[33] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.
[34] N. Tsantalis, A. S. Ketkar, and D. Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* PP, 99 (2020), 1–1.
[35] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. 2009. Design rule hierarchies and parallelism in software development tasks. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 197–208.
[36] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. Gems: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 24–34.
[37] Yutong Zhao, Lu Xiao, Xiao Wang, Zhifei Chen, Bihuan Chen, and Yang Liu. 2020. Butterfly space: An architectural approach for investigating performance issues. In *2020 IEEE International Conference on Software Architecture (ICSA).* IEEE, 202–213.

Received 2024-07-05; accepted 2024-07-26