



One-to-One or One-to-Many? Suggesting Extract Class Refactoring Opportunities with Intra-class Dependency Hypergraph Neural Network

Di Cui

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

Qiangqiang Wang

Xidian University
Xi'an, China
22031212112@stu.xidian.edu.cn

Yutong Zhao

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

Jiaqi Wang

Xidian University
Xi'an, China
23031212446@stu.xidian.edu.cn

Minjie Wei

Xidian University
Xi'an, China
23031212380@stu.xidian.edu.cn

Jingzhao Hu

Xidian University
Xi'an, China
hujingzhao@xidian.edu.cn

Luqiao Wang

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

Qingshan Li

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

Abstract

Excessively large classes that encapsulate multiple responsibilities are challenging to comprehend and maintain. Addressing this issue, several Extract Class refactoring tools have been proposed, employing a two-phase process: identifying suitable fields or methods for extraction, and implementing the mechanics of refactoring. These tools traditionally generate an intra-class dependency graph to analyze the class structure, applying hard-coded rules based on this graph to unearth refactoring opportunities. Yet, the graph-based approach predominantly illuminates direct, “one-to-one” relationship between pairwise entities. Such a perspective is restrictive as it overlooks the complex, “one-to-many” dependencies among multiple entities that are prevalent in real-world classes. This narrow focus can lead to refactoring suggestions that may diverge from developers’ actual needs, given their multifaceted nature. To bridge this gap, our paper leverages the concept of intra-class dependency hypergraph to model *one-to-many* dependency relationship and proposes a hypergraph learning-based approach to suggest Extract Class refactoring opportunities named HECS. For each target class, we first construct its intra-class dependency hypergraph and assign attributes to nodes with a pre-trained code model. All the attributed hypergraphs are fed into an enhanced hypergraph neural network for training. Utilizing this trained neural network alongside a large

language model (LLM), we construct a refactoring suggestion system. We trained HECS on a large-scale dataset and evaluated it on two real-world datasets. The results show that demonstrates an increase of 38.5% in precision, 9.7% in recall, and 44.4% in f1-measure compared to 3 state-of-the-art refactoring tools including JDeodorant, SSECS, and LLMRefactor, which is more useful for 64% of participants. The results also unveil practical suggestions and new insights that benefit existing extract-related refactoring techniques.

CCS Concepts

• Software and its engineering → Software Maintenance.

Keywords

Extract Class Refactoring, Hypergraph Neural Network

ACM Reference Format:

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 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, September 16–20, 2024, Vienna, Austria. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3650212.3680379>

*Qingshan Li is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0612-7/24/09

<https://doi.org/10.1145/3650212.3680379>

1 Introduction

Excessively large classes that encapsulate multiple responsibilities within a single class are challenging to comprehend and maintain. [23] To mitigate this issue [39, 49], developers use an Extract Class refactoring operation to decompose large classes into smaller ones and generate new classes to be reused. This refactoring involves two phases: (i) suggesting proper fields or methods to extract and (ii) applying mechanics to perform refactoring operation, which improves the class’s internal structure without changing the external behaviours [52].

Several automatic Extract Class refactoring tools have been proposed [6, 7, 12, 13, 30, 41, 50, 60]. Most of these tools [7, 12, 13, 30] first extract intra-class dependency graph, where nodes represent fields or methods within class and edges represent field-access or method-call dependencies. They further quantify heuristic rules based on this graph to suggest refactoring opportunities. However, in most cases, the intra-class dependency graph merely focuses on the *one-to-one* dependency relationship between two entities and fails to capture the *one-to-many* dependency relationship among multiple entities, which can be challenging to represent rich extraction cases of multiple entities in practice. This may result in the suggested refactoring opportunities do not align with developers' preferences.

To more comprehensively capture the dependency relationships and enhance the accuracy of our refactoring suggestions, we introduce the concept of hypergraph. This advanced representation allows for edges that bind multiple nodes, thus reflecting the reality of code where multiple fields and methods often interact in unison. The hypergraph paradigm, with its ability to embody a cohesive group of fields/methods, sets the stage for more precise and encompassing refactoring opportunity suggestions. Building on the robustness of hypergraph theory, we propose the **Hypergraph learning-based Extract Class refactoring detection System** that elevates the Extract Class refactoring process. HECS utilizes historical samples to mine for patterns and dependencies, thereby automating the detection and suggestion of refactoring opportunities with heightened relevance and accuracy.

The workflow of our HECS approach is outlined as follows: Initially, we extract intra-class dependency graphs from both training and testing samples and transform these into hypergraphs. Subsequently, nodes in these hypergraphs are assigned with attributes with pre-trained code model and these hypergraphs are fed into an enhanced hypergraph neural network for the purpose of training. Ultimately, utilizing the trained neural network alongside a large language model (LLM), we construct a refactoring suggestion system. This system aids in the identification and extraction of appropriate fields and methods into a new class. HECS underwent training on a large-scale dataset [10] and was evaluated against two real-world datasets [32, 53]. This evaluation included a systematic comparison with three cutting-edge tools: JDeodorant [13], SSECS [12], and LLMRefactor [48], which have been proven to be effective in previous work [18, 49]. The results suggest that HECS surpasses these state-of-the-art tools in both effectiveness and usefulness. Additionally, the results reveal insightful and practical implications of hypergraph utilization, benefiting further extract-related refactoring approaches such as Extract Variable and Extract Method.

In summary, we make the following contributions:

- A new perspective of hypergraph to suggest Extract Class refactoring opportunities.
- A systematic exploration of implementations of HECS on combinations of 6 pre-trained code models and 2 hypergraph neural networks. The experimental results reveal that the following combinations achieve the best results: 1) HGNN+CoT-exT, and 2) HGNN+CodeBERT.

- A comprehensive evaluation of HECS on two real-world datasets. HECS demonstrates an increase of 38.5% in precision, 9.7% in recall, and 44.4% in f1-measure compared with the best results of state-of-the-art tools, including JDeodorant [13], SSECS [12], and LLMRefactor [48]. The questionnaire results also indicate that HECS is more useful than state-of-the-art tools for 64% of participants.
- A benchmark to investigate the effectiveness of hypergraph neural networks in suggesting Extract Class refactoring opportunities. All the data are publicly available [3].

2 Background and Motivation

2.1 Problem Definition

Extract Class Refactoring Opportunities Suggestion. Let $\mathcal{F}\mathcal{M}_{tc} = \{\{F_1, F_2, \dots, F_n\}, \{M_1, M_2, \dots, M_m\}\}$ be a set of fields and methods of the target class: tc , where n and m are the numbers of fields and methods. The Extract Class refactoring opportunities suggestion can be regarded as discovering movable fields and methods as a set of extracted classes:

$$ECSet = \{EC_i | i = 1, \dots, r\} \quad (1)$$

where r represents the number of extracted classes. Each extracted class EC_i can be formally defined as follows:

$$EC_i = \{\{F_j | j = 1, 2, \dots, k \wedge F_j \in \mathcal{F}\mathcal{M}_{tc} \wedge k < n\}, \quad (2)$$

$$\{M_j | j = 1, 2, \dots, s \wedge M_j \in \mathcal{F}\mathcal{M}_{tc} \wedge s < m\}\} \quad (3)$$

where k and s represent the numbers of fields and methods in the extracted class EC_i respectively. Note that, each field or method in target class: tc only belongs to one extracted class. Fig. 1 presents the general process of suggesting Extract Class refactoring opportunities.

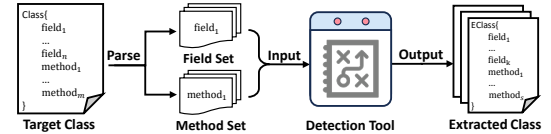


Figure 1. General process of suggestion of Extract Class refactoring opportunities.

Intra-class Dependency Graph. Given a class, the Intra-class Dependency Graph is defined as: $G_c = (V_c, E_c)$, where V_c represents fields or methods within class and E_c represents field-access dependencies (field read and field write) or method-call (invocation of methods or constructors) dependencies among nodes within class.

Intra-class Dependency Hypergraph. An Intra-class Dependency Hypergraph is the hypergraph format of Intra-class Dependency Graph, which is defined as $\mathcal{G}_c = (V_c, \mathcal{E}_c, \mathbf{W}_c)$ with a diagonal matrix \mathbf{W}_c representing weights for each hyperedge. By default, each hyperedge is assigned a weight of 1.0. V_c is a vertex set, which is identical to V_c . \mathcal{E}_c represents the hyperedges constructed from E_c , which will be illustrated in Section III.A. Each hyperedge includes a group of vertices adjacent to each other. Thus, the hypergraph: \mathcal{G}_c can be denoted by a $|V_c| \times |\mathcal{E}_c|$ incidence matrix \mathbf{H}_c , where entries can be defined as follows:

$$h(v, e) = \begin{cases} 1 & \text{if } v \in e, \\ 0 & \text{if } v \notin e. \end{cases} \quad (4)$$

For a vertex $v \in \mathcal{V}_c$, its degree is defined as $d(v) = \sum_{e \in \mathcal{E}_c} \mathbf{H}_c(v, e)$. For a hyperedge $e \in \mathcal{E}_c$, its edge degree is defined as $\delta(e) = \sum_{v \in \mathcal{V}_c} \mathbf{H}_c(v, e)$. \mathbf{D}_v and \mathbf{D}_e denote the diagonal matrices of vertex degrees and edge degrees, respectively.

2.2 Motivation Example

We use a real-world example of commit dc38ef3 [20] in an open-source project: Xerces [51] to illustrate the Extract Class Refactoring operation, which is demonstrated in Fig. 2. Fig. 2(a) presents the code snippet of the target class: `IDDataTypeValidator` including 9 fields and 6 methods, which is utilized to validate ID data type. Lines 5 to 8 and 95 to 97 highlighted with red colour and “-” marks, can further be extracted as the basis to create a separate class: `HexBinaryDatatypeValidator` for these fields and methods are related to the validation of hex data type. Fig. 2(b) presents the code snippet of extracted class: `HexBinaryDatatypeValidator` including 8 fields and 7 methods. Class: `HexBinaryDatatypeValidator` is extracted from class: `IDDataTypeValidator` and added with new fields and methods. In Fig. 2(b), we highlight the extracted fields or methods with green colour and “+” marks. Line 2, Line 3, Line 8, and Line 9 map the extracted fields: `fLocale`, `fLength`, `fEnumeration`, and `fFacetsDefined` from Fig. 2(a) in turn. Lines 20 to 22 map the extracted method: `setLocale()` from Fig. 2(a).

For the code snippet in Fig. 2(a), Fig. 2(c) and Fig. 2(d) present the corresponding intra-class dependency graph and intra-class dependency hypergraph respectively, the construction detail be illustrated in Section III. Nodes with dark grey color and “*” mark represent extracted fields or methods. Regarding as intra-class dependency graph and intra-class dependency hypergraph, from their graph and matrix representation, we observed that these two types of graph present a significant difference. For example, the total degree of vertices in intra-class dependency graph is 22 while the total degree of vertices in intra-class dependency hypergraph is 15. It is intuitive that their effectiveness may also be varied when predicting refactoring candidates. In our paper, we will fully exploit intra-class dependency hypergraph and thus propose a hypergraph-learning based approach: HECS for suggesting Extract Class refactoring opportunities.

3 Methodology

Fig. 3 presents an overview of the HECS pipeline that aims to suggest Extract Class refactoring candidates automatically. HECS operates in two phases: training phase and detection phase. The training phase includes hypergraph construction, node attribute generation, and hypergraph learning, and finally returns a well-trained model for refactoring opportunity suggestion. The detection phase first constructs attributed hypergraph through hypergraph construction and node attribute generation. Based on constructed attributed hypergraphs, the detection phase further conducts refactoring opportunity suggestion via hierarchical model invocation and LLM-based pre-condition verification.

3.1 Hypergraph Construction

Given a program as input, we first extract intra-class dependency graph using *ENRE* [31], a state-of-the-art static analysis tool for extraction of code dependencies. *ENRE* supports more than 11 types

of code dependencies and provides flexible interfaces to implement the customized analyzer in a concise manner.

Inspired by design rule space which models code dependency graph as multiple overlapping spaces [56], we devise a heuristic algorithm to discover all the design spaces of intra-class dependency graph and convert them into hypergraphs. Alg. 1 presents the procedure. The input is the intra-class dependency graph: G_c comprising vertex set: V_c and edge set: E_c . The output is intra-class dependency hypergraph: \mathcal{G}_c . Line 1 first calculates the transitive closure of E_c as E_c^* . For each vertex: v_i in V_c , a hyperedge he_i is constructed by including the vertex itself and all reachable adjacent vertices connected to v_i through edges in E_c^* (Lines 4-8). These hyperedges are then added to the intra-class dependency hypergraph \mathcal{G}_c (Line 9). Line 11 finally returns the well-constructed hypergraph: \mathcal{G}_c .

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

```

1:  $E_c^* \leftarrow \text{TransC}(E_c)$  ▷ The transitive closure of  $E_c$ 
2: for each  $v_i$  in  $V_c$  do
3:    $he_i \leftarrow v_i$ 
4:   for each  $v_j$  in  $V_c$  do
5:     if  $(v_j, v_i) \in E_c^*$  then
6:        $he_i.append(v_j)$ 
7:     end if
8:   end for
9:    $\mathcal{G}_c.add(he_i)$  ▷ Construct the hyperedge for each vertex
10: end for
11: return  $\mathcal{G}_c$ 

```

3.2 Node Attribute Generation

Suppose the vertex set of intra-class dependency hypergraph \mathcal{G}_c is: $\{v_i | i = 1, 2, \dots, n \wedge v_i \in \mathcal{V}_c\}$. For each vertex: $v_i \in \mathcal{V}_c$, we generate its attribute from code snippets with pre-trained code model. Pre-trained code model is trained on massive corpora and learns a general-purpose code representation. This paper focuses on 6 representative pre-trained code models: CodeBERT, GraphCodeBERT, CodeGPT, CodeT5, CoTexT, and PLBART, frequently used in previous literature [58], which are illustrated in Table 1. We use line of code as the basic unit and split the field/method as a set of lines of code: $\{c_i | i = 1, 2, \dots, k\}$. The corresponding embedding result generated by the pre-trained code model can be: $\{ebd(c_i) | i = 1, 2, \dots, k\}$. we use the mean-pooling to aggregate embedding results of k lines of code as node attribute: $x_i \in \mathbb{R}^d$:

$$x_i = \text{Mean}(\{ebd(c_1), \dots, ebd(c_k)\}) \quad (5)$$

In our paper, we followed previous work [46] and heuristically set d for 768. We obtain the node attribute space of \mathcal{G}_c as: $\mathbf{X}_c = \{x_i | i = 1, 2, \dots, n\}$, which will be provided as the input of hypergraph learning.

3.3 Hypergraph Learning

Hyperlink Prediction. Suggestion of Extract Refactoring opportunities can be modeled as a hyperlink prediction problem, aiming to find the most likely existent hyperlinks missing from the observed hyperedge set: \mathcal{E}_l . For a given potential hyperlink: l , most hyperlink

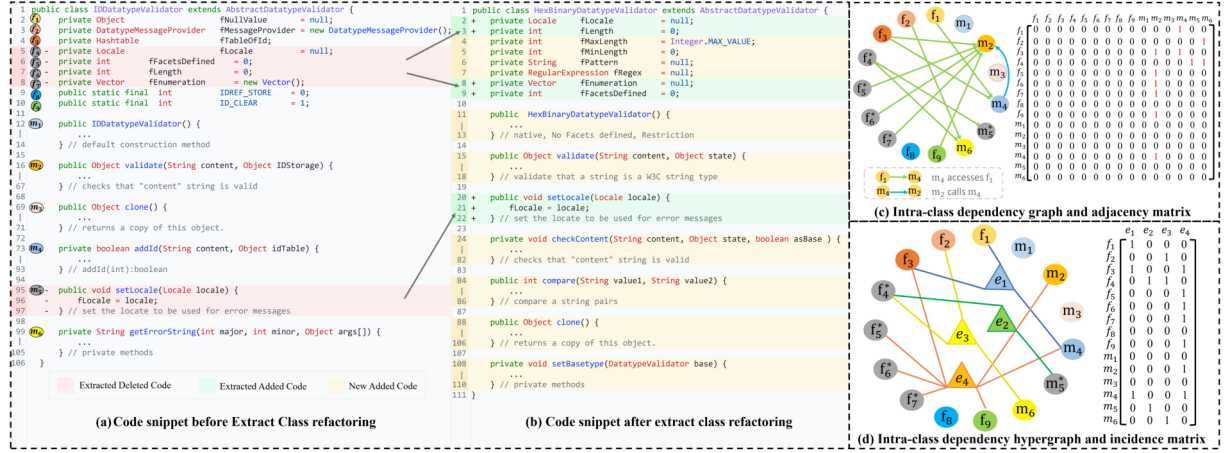


Figure 2. An illustrated motivation example of Extract Class Refactoring operation. (Lines 5 to 8 and 95 to 97 in (a) are extracted as the class: `HexBinaryDatatypeValidator` in (b), nodes with dark grey color and “*” mark represent extracted fields or methods)

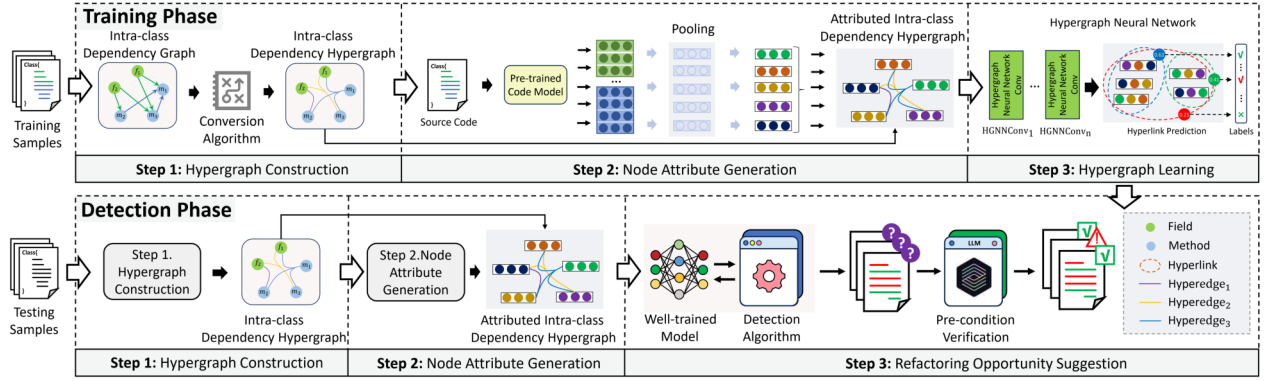


Figure 3. Overview of the proposed approach: HECS including training phase and detection phase.

Table 1: Characteristics of selective pre-trained code models.

Model	Size	Architecture	Pre-trained Dataset
CodeBERT [22]	125M	RoBERTa	CodeSearchNet [29]
GraphCodeBERT [26]	125M	RoBERTa	CodeSearchNet [29]
CodeGPT [34]	124M	GPT2	CodeSearchNet [29]
CodeT5 [55]	220M	T5	CodeSearchNet [29] and BigQuery [25]
CoText [45]	220M	T5	CodeSearchNet [29] and BigQuery [25]
PLBART [5]	140M	BART	Self-Collected

prediction methods aim to learn a function Ψ such that:

$$\Psi(l) = \begin{cases} \geq \epsilon & \text{if } l \in \mathcal{E}_l, \\ < \epsilon & \text{if } l \notin \mathcal{E}_l, \end{cases} \quad (6)$$

where ϵ is a threshold to binarize the continuous value of Ψ into a label. In this paper, we heuristically set ϵ to 0.5 and proposed a hypergraph neural network-based method for hyperlink prediction to suggest Extract Class refactoring opportunities.

Hypergraph Neural Network. Hypergraph neural network is a deep learning model for hypergraph data, which can learn node representations through multiple convolution layers. The initial hypergraph neural network (HGNN) [21] performs feature smoothing

on nodes through each convolution layer as follows:

$$\mathbf{X}^{t+1} = \sigma(\mathbf{L}\mathbf{X}^t\Theta^t) \quad (7)$$

$$\mathbf{L} = \mathbf{D}_v^{-\frac{1}{2}}\mathbf{H}\mathbf{W}_e\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-\frac{1}{2}}$$

where $\sigma(\cdot)$ is a non-linear activation function. \mathbf{X}^t is the input vertex feature matrix of layer t . Θ^t is the learnable parameters of layer t . \mathbf{L} is the Laplacian matrix of the hypergraph incidence matrix: \mathbf{H} . \mathbf{W}_e is a diagonal hyperedge weight matrix. \mathbf{D}_v is a diagonal vertex degree matrix. \mathbf{D}_e is a diagonal hyperedge degree matrix.

In this paper, we employ the improved version of HGNN: HGNN⁺ [24], which enhances the initial HGNN by concatenating on hyperedge group-level information fusion compared with hyperedge-by-hyperedge in HGNN. The convolution layer of HGNN⁺ (named HGNNConv⁺) can be defined as:

$$\mathbf{X}^{t+1} = \sigma(\mathbf{D}_v^{-1}\mathbf{H}\mathbf{W}_e\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{X}^t\Theta^t) \quad (8)$$

Specifically, each convolution layer HGNNConv⁺ fuses node embeddings of hypergraph with a two-stage message passing process: 1) aggregating node messages to hyperedges, and 2) aggregating

hyperedge messages back to nodes. We heuristically set the convolution layers to 2 in our paper.

HGNN-based Hyperlink Prediction. Fig. 4 presents the overview of our HGNN-based hyperlink prediction framework including 6 layers, which is illustrated as follows:

Input Layer: this layer takes hyperlink prediction matrix: \mathcal{E}_l and the attributed intra-class dependency hypergraph \mathcal{G}_c including a node attribute space: \mathbf{X}_c and an incidence matrix: \mathbf{H}_c as input.

Hypergraph Neural Network Layer: This layer updates the node attribute space: \mathbf{X}_c of hypergraph: \mathcal{G}_c through two layers of HGNN-Conv⁺ in Eq. (8) to produce an updated node attribute space: \mathbf{X}'_c .

Dropout Layer: This layer temporarily discards 10% of hidden neurons to prevent over-fitting, which improves model's generalization ability.

Pooling Layer: This layer uses the mean-pooling to aggregate node attribute space: \mathbf{X}'_c and calculate the hyperedge attribute space: \mathbf{X}'_l of hyperlink prediction matrix: \mathcal{E}_l . For each hyperlink: $l \in \mathcal{E}_l$, the corresponding embedding: x'_l is computed as follows:

$$x'_l = \text{Mean}(\{x'_{v_i} \mid v_i \in l \wedge x'_{v_i} \in \mathbf{X}'_c\}) \quad (9)$$

where *Mean* is the mean-pooling operation.

Fully Connected Layer and Output Layer: The fully connected layer converts the output of the pooling layer into a one-dimensional vector as hyperlink prediction results with activation function: *sigmoid*. For each hyperlink: l , the predicted probability result: p_l is computed as follows:

$$p_l = \text{sigmoid}(W^c \cdot x'_l + b) \quad (10)$$

where W^c and b represent the weight parameter and bias term respectively. p_l represents the probability that the hyperlink: l is predicted as positive. The cross-entropy loss function: $\mathcal{L}_{\text{hyperlink}}$ is used to train the model, which is defined as follows:

$$\mathcal{L}_{\text{hyperlink}} = \frac{1}{N} \sum_l -[y_l \cdot \log(p_l) + (1 - y_l) \cdot \log(1 - p_l)] \quad (11)$$

where y_l represents the real label of hyperlink: l (1 for positive and 0 for negative).

3.4 Refactoring Opportunity Suggestion

Given a target class as input, we first extract its attributed intra-class dependency hypergraph and feed it into a well-trained model and generate a group of fields and methods as candidates. For each candidate, we use LLMs to verify its pre-conditions and return the final results.

Trained Model Invocation: Inspired by hierarchical clustering algorithm [40], we devised a heuristic algorithm invoking the well-trained model to generate refactoring candidates presented in Alg. 2. Given a target class: tc , this algorithm takes the attributed hypergraph: \mathcal{AG}_{tc} , the set of fields/methods: \mathcal{FM}_{tc} , and the number of extracted classes: r as input. Line 1 first initializes each field/method as an individual cluster. Line 2 uses the matrix: \mathbf{P} to store the likelihood to be merged between clusters and initializes as \emptyset . Lines 3 to 5 generate all the cluster pairs and calculate the likelihood score with the well-trained model. Lines 6 to 17 iteratively merge clusters until the number of extracted classes is satisfied. Line 7 obtains the cluster pair: $(clus_i, clus_j)$ with the highest score: $score_h$ in matrix: \mathbf{P} . If $score_h$ is less than 0.5, it indicates that no classes

can be extracted. Line 11 merges the cluster pair: $(clus_i, clus_j)$ into a new cluster: $clus_{new}$. Lines 12 and 16 remove $clus_i$ and $clus_j$ and add $clus_{new}$ to the *clusters* respectively. Lines 13 to 15 update the matrix: \mathbf{P} with model. Line 18 collects *clusters* as Extracted Class refactoring candidates.

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: $\mathcal{FM}_{tc} = \{fm_i \mid i = 1, 2, \dots, n\}$ - Fields/methods of tc
Input: r - Number of extracted classes
Output: $ECSet$ - Set of extracted classes

```

1: clusters  $\leftarrow \{\{fm_i\} \mid i = 1, 2, \dots, n\}$  ▷ Initialize clusters
2:  $\mathbf{P} \leftarrow \emptyset$  ▷ Initialize proximity adjacent matrix
3: for all pair in AllPair(clusters) do
4:    $\mathbf{P}[\text{pair}[0]][\text{pair}[1]] \leftarrow \text{Model}(\mathcal{AG}_{tc}, \text{pair})$ 
5: end for
6: while  $|\text{clusters}| > r$  do
7:    $score_h, (clus_i, clus_j) \leftarrow \text{MaxValue}(\mathbf{P}, \text{clusters})$ 
8:   if  $score_h < 0.5$  then
9:     return  $\emptyset$  ▷ No classes can be extracted
10:  end if
11:   $clus_{new} \leftarrow \text{Merge}(clus_i, clus_j)$ 
12:   $\text{clusters} \leftarrow \text{clusters} - \{clus_i\} - \{clus_j\}$ 
13:  for all  $clus_k$  in clusters do
14:     $\mathbf{P}[clus_k][clus_{new}] \leftarrow \text{Model}(\mathcal{AG}_{tc}, (clus_{new}, clus_k))$ 
15:  end for
16:   $\text{clusters} \leftarrow \text{clusters} + \{clus_{new}\}$ 
17: end while
18:  $ECSet \leftarrow \text{clusters}$ 
19: return  $ECSet$ 
```

Fig. 5 presents the procedure of refactoring opportunity suggestion algorithm with hypothetical target class comprising 3 fields, and 4 methods. Fig. 5 presents the procedure of 5 iterations and finally returns 2 Extracted Class candidates: (f_1, m_2, m_3, m_4) and (f_2, f_3, m_1) .

LLM-based Pre-Condition Verification. Refactoring operation may frequently introduce bugs and violate the consistency of program semantic [11, 19]. Following the work of Tsantalis et al. [4, 47], given an Extract Class refactoring candidate, these pre-conditions before refactoring implementation should be formally verified to guarantee the behaviour preservation and functional usefulness of refactored program, which are listed as follows:

- 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.
- 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.

Previous work [4, 11, 19, 35, 36] use heuristic rules/algorithms for static code analysis to verify above pre-conditions. In most cases, these approaches rely on expert knowledge, which can be

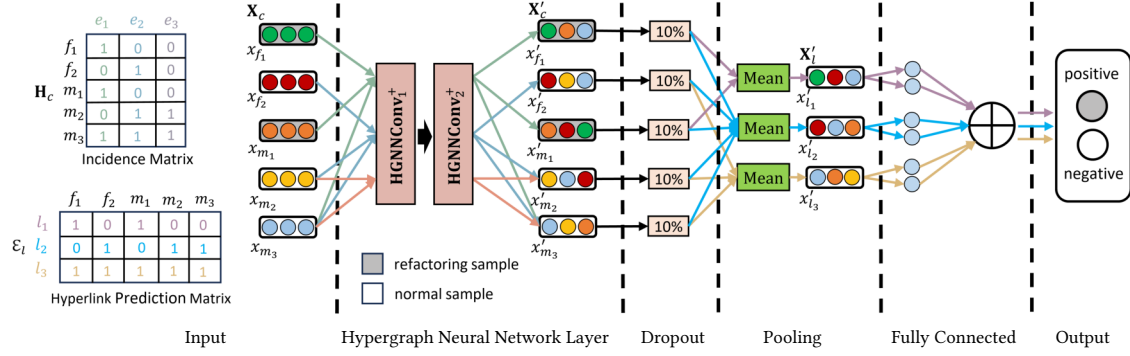


Figure 4. Overview of HGNN-based hyperlink prediction framework.

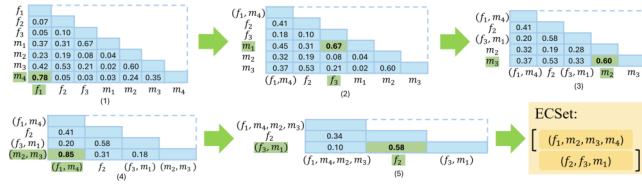


Figure 5. An illustrated example of the procedure of refactoring opportunity suggestion algorithm.

challenging to match all the extreme cases, capture implicit code references and implement an accurate pre-condition checker. In our paper, we use a large language model-based approach, which can adapt to various cases with prompt engineering. Therefore, we utilize two-step prompt engineering to harness LLM’s knowledge for pre-condition verification. In the Step 1, we enable the LLM to extract pre-condition relevant fields/methods through few-shot learning. Fig. 7 uses the pre-condition: **P5** as a running example to present the used extraction prompt. In the Step 2, we formulate verification prompt for fine-tuned LLM to check whether relevant entities violate pre-conditions. Fig. 8 presents a running example to illustrate the used verification prompt. In our paper, we use GPT-3.5 [42] and heuristically use 3 instances for each pre-condition in few-shot learning. All the used prompts and instances are available [3]. For each refactoring candidate, we filter out invalid candidates through step2 prompt engineering with LLM to obtain the final suggestion results.

3.5 Prototype Implementation

As presented in detection view (Fig. 6.(a)), users first click the selection button to select the target class and then click the “detection button” (Button 1) to obtain suggestion results, which are highlighted with red colour in code editor. Furthermore, all the suggestion results will be summarized in an Extraction Summary Table, where column presents the name, type, and range of number of lines of code. When users click “preview” button (Button 2), a refactoring view shown in Fig. 6.(b) will be demonstrated, which previews suggested results in the form of code differences. The added code and deleted code are highlighted with the green color and red color respectively. After reviewing all the suggestion results, users can click the “refactoring” button (Button 3) to execute Extract Class refactoring operations automatically.

4 Evaluation

The evaluation section aims to address the three research questions as follows:

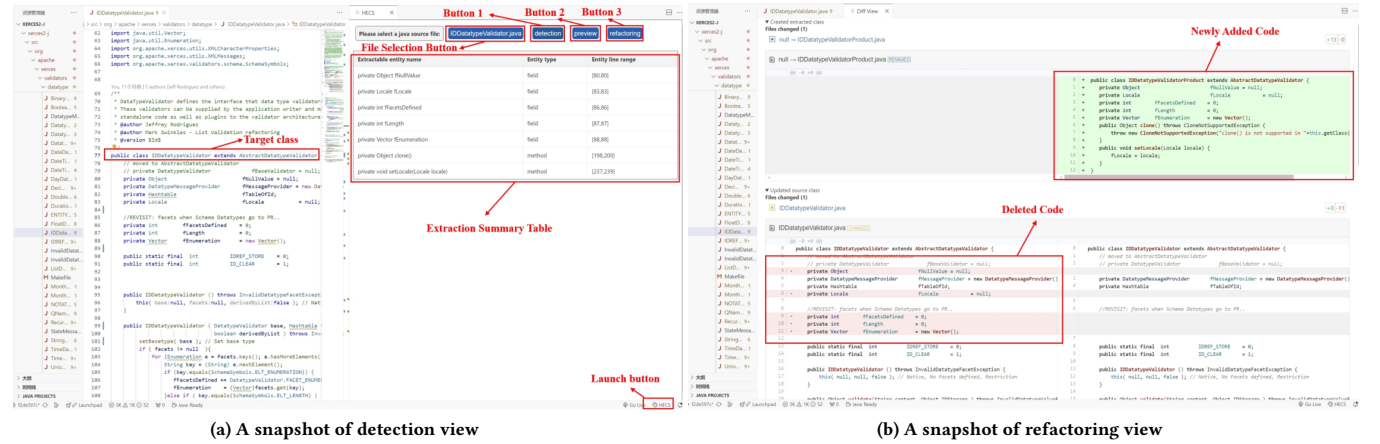
- RQ1: Which hypergraph neural networks are the most effective in suggesting Extract Class refactoring opportunities?** The answer to this question would help us better understand the impact of attributed intra-class property hypergraph on the performance of HECS.
- RQ2: How accurately does HECS suggest Extract Class refactoring opportunities compared to state-of-the-art refactoring tools?** The answer to this question would demonstrate how well HECS performs in terms of accuracy compared to state-of-the-art refactoring tools.
- RQ3: How useful does HECS suggest Extract Class refactoring opportunities for developers?** The answer to this question would shed insight into the usefulness of HECS in real-world applications.

4.1 Experiment Setup

We illustrated the used datasets, evaluation metrics, and experiment settings as follows:

Datasets. We train HECS on the most large-scale refactoring dataset: *ECTrainSet* (RQ1) based on the best of our knowledge, and evaluate the accuracy of HECS with a quite high-quality dataset annotated by experts: *ECAccEval* (RQ2) and the usefulness of HECS with a hand-written dataset with a group of typical cases: *ECHumanEval* (RQ3), which is frequently used in the user study of previous work [6, 9, 12, 14, 30]. Statistics of used datasets are presented in Table 2, including involved projects (#Project), the number of extracted classes (#EClass), the number of involved extracted methods (#EMethod), and the number of involved extracted fields (#EField).

ECTrainSet [10] is crawled from 11,149 projects of 3 communities: GitHub, F-Droid, and Apache, containing 41,191 Extract Class refactoring operations. We use *ECTrainSet* for model training and believe it is comprehensive and diverse. *ECAccEval* [53] contains 106 real-world Extract Class refactoring oracles from 185 projects and each instance is rigorously verified by more than two experts, which is used for accuracy evaluation. *ECHumanEval* [32] is collected from two medium-sized projects GanttProject [1] and Xerces [51] including 14 typical Extract Class refactoring cases, which



(a) A snapshot of detection view

(b) A snapshot of refactoring view

Figure 6. The graphical user interface of our prototype implementation as a VSCode extension.

Input Prompt:

Consider the task of extracting the pre-condition: **P5** relevant entities from the test code snippet (along with three examples):

1. Pre-condition: **P5** refers to the field that is accessed by fields or methods of other classes.
2. Few-shot Learning Examples 1-3:
[code snippet1] [example input1] [example output1]
...
3. Consider the following test code snippet:

```

class B {
    private int f1;
    public void m1(int val) { f1 = val; }
}
class C {
    private B b;
    public C() { b = new B(); }
    public void m1() { b.m1(10); }
}

```

Does the code snippet have pre-condition: **P5** relevant entities?

Completions:

- yes, B.f1

Figure 7. A running example of the used extraction prompt.

Input Prompt:

1. Pre-condition rules (**P1-P8**) as follows:

...

2. Consider the following test extract class candidate:

Source class:

```

class A {
    int f1;
    double f2;
    void m1(){...}
    int m2(){ synchronized (this){...} }
}

```

Extract class candidate:

```

- field: A.f2
- method: A.m2()

```

Please verify whether the candidate meet pre-conditions **P1-P8** by extracting the pre-condition relevant entities from the candidate and presenting the verification results in tabular form.

Completions:

Pre-conditions	P1	P2	P3	P4	P5	P6	P7	P8	RESULT
Extract Entity								A.m2() A.f2,A.m2()	
Verification								x	FAILED

Figure 8. A running example of the used verification prompt.

Table 2: Statistics of used datasets.

Dataset	#Project	#EClass	#EMethod	#EField
<i>ECTrainSet</i> [10]	11149	41191	224404	24043
<i>ECAccEval</i> [53]	185	106	554	53
<i>ECHumanEval</i> [32]	2	19	66	72

have been continuously investigated by researchers for usefulness evaluation since 2009.

Evaluation Metrics. Following previous refactoring-related work [57], we employ three frequently used evaluation metrics: Precision, Recall, and F1-Measure to evaluate HECS, which are defined as follows:

$$\text{Precision} = \frac{\# \text{ of correct suggested refactorings}}{\# \text{ of suggested refactorings}} \quad (12)$$

$$\text{Recall} = \frac{\# \text{ of correct suggested refactorings}}{\# \text{ of correct refactorings}} \quad (13)$$

$$\text{F1-Measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (14)$$

Specifically for the extraction of multiple classes, we first construct a weighted complete bipartite graph that comprises extracted classes of ground-truth and detection results. The weight of each edge is the number of overlapped fields or methods. We further use the Hungarian algorithm [38] to find the maximum weighted matching in this bipartite graph and calculate the above evaluation metrics based on this matching.

Experiment Settings. We conducted all the experiments on a 2.4GHz Intel Xeon-4210R server with 10 logical cores, 128GB of memory, and 4 NVIDIA RTX 4090 GPUs. We followed most of the default hyper-parameters of pre-trained models in previous work [16]. We implemented all the neural networks based on the Python library: PyTorch [44].

To address RQ1 (combination evaluation), we implement HECS with 2 hypergraph neural networks and 6 pre-trained code models on *ECTrainSet* [10], and exhaustively compare their performance on *ECTrainSet*. To minimize the impact of experimental randomness,

we repeated the 10-fold cross-validation 10 times (10×10) and computed the average value of evaluation metrics as the final results. We use hypersmote [8] to address the imbalance issue.

To address RQ2 (accuracy evaluation), we select the best combinations of hypergraph neural network and pre-trained code model according to the results of RQ1 and evaluate their performance on *ECAccEval* [53] and compare these results to 3 state-of-the-art Extract Class refactoring tools including LLMRefactor [48], SSECS [12], and JDeodorant [13]. JDeodorant and SSECS have been proven to be the most effective tools in previous work [49]. We additionally include LLMRefactor, a representative LLM-based refactoring tool that has recently emerged, for comparison. LLMRefactor is the most relevant refactoring tool in LLM4SE direction although it is not specialized in extracting class. We configure LLMRefactor with the default prompt and examples. We configure SSECS and JDeodorant with the default parameters. We contacted the authors with extensive conversations to fully understand how to use these tools and replicate their results best. All the configuration details are also available [3].

To address RQ3 (usefulness evaluation), we conducted a user study by hiring 50 software engineers with more than 6 years of industrial experience. All the participants are not the authors of this paper. They are invited to analyze the suggested results of 4 refactoring tools including the best combination of HECS in RQ1 and RQ2, LLMRefactor [48], SSECS [12], and JDeodorant [13] on *ECHumanEval* [32]. We further encourage participants to complete a questionnaire regarding their practical experiences with each refactoring tool.

4.2 Experiment Result

Combination Analysis (RQ1). We conduct a systematic comparison of HECS on 12 combinations of 6 pre-trained code models and 2 hypergraph neural networks. We further conduct the controlled experiment of hypergraph neural network and graph neural network. Within the framework of HECS, we utilize the intra-class dependency graph and train three graph neural networks: GraphSAGE, GCN, and GAT with the assistance of GraphSMOTE [59] for comparison. Table 3 presents the performance of various combinations on the *ECTrainSet* [10], where rows and columns are labeled by pre-trained code model and hypergraph/graph neural network respectively. For each hypergraph/graph neural network, we highlight the greatest precision, recall, and f1-measure scores using a gray background color. Furthermore, we highlight the greatest scores of all possible combinations using a “*” mark. As presented in Table 3, we observed that combinations of HGNN+CoTexT and HGNN⁺+CodeBERT outperform other combinations. HGNN+CoTexT achieves the highest precision score, whereas HGNN⁺+CodeBERT achieves the highest recall and f1-measure score. We further compare the performance of hypergraph neural networks and graph neural networks on various pre-trained code models. As presented in Table 3, we observed that hypergraph neural networks: HGNN and HGNN⁺ outperform graph neural networks: GraphSAGE, GCN, and GAT on most pre-trained code models. The reverse is true in few cases. For example, GCN outperforms HGNN⁺ in precision score on most pre-trained code models. A possible explanation is

that these pre-trained code models may generate features that are easier for graph neural networks to capture and thus perform better.

To verify the significance of the hypergraph/graph neural network on experimental results, we further performed effect size: Cliff’s δ [15], a non-parametric effect size, to check whether a statistically significant difference exists between results. The values range from negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), and large ($|\delta| \geq 0.474$). We present statistical test results of hypergraph/graph neural networks in Table 4. As presented in Table 4, we observed a significant difference in precision, recall, and f1-measure between most hypergraph/graph neural networks.

Answer to RQ1: Hypergraph neural networks outperform graph neural networks in suggesting Extract Class refactoring opportunities. The most effective combinations of hypergraph neural network and pre-trained code model include HGNN+CoTexT and HGNN⁺+CodeBERT.

Accuracy Evaluation (RQ2). To evaluate the accuracy of HECS, we selected the two most effective combinations: HGNN+CoTexT and HGNN⁺+CodeBERT in RQ1. We regard these two combinations as two variants of HECS: HECS- α and HECS- β , corresponding to HGNN+CoTexT and HGNN⁺+CodeBERT respectively. Furthermore, we apply trained neural networks on these combinations and evaluate their performance on *ECAccEval* [53]. We compare HECS with three state-of-the-art Extract Class refactoring tools including LLMRefactor [48], SSECS [12], and JDeodorant [13].

Table 5 presents the average accuracy of each refactoring tool on each instance of *ECAccEval* [53]. We highlight the greatest precision, recall, and f1-measure score of LLMRefactor, JDeodorant, and SSECS with red color. We also highlight the greatest score of HECS- α and HECS- β with blue color. As presented in Table 5, 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 the f1-measure score. We use the recall score of LLMRefactor, the precision score of SSECS, and the f1-measure score of JDeodorant as baseline. 2) The best results of HECS: HECS- β 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. HECS- α presents a lower recall score than LLMRefactor and JDeodorant but a higher f1-measure score. The improvement of HECS can be attributed to effective capture of refactoring characteristics with hypergraph neural network. Software practitioners can consider HECS- β as a reliable refactoring tool to suggest Extract Class refactoring opportunities. We will select HECS- β for usefulness evaluation in RQ3.

Answer to RQ2: HECS demonstrates an increase of 38.5% in precision, 9.7% in recall, and 44.4% in f1-measure compared to state-of-the-art Extract Class refactoring tools.

Usefulness Evaluation (RQ3). To evaluate the usefulness of HECS, we conducted a user study of 50 industrial engineers to

Table 3: Performance of various combinations on the *ECTrainSet* [10].

Pre-trained Model	HGNN [21]			HGNN+ [24]			GraphSAGE [28]			GCN [33]			GAT [54]		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
CodeBERT [22]	.926	.584	.716	.705	.753*	.728*	.597	.596	.597	.790	.334	.469	.654	.592	.622
CodeT5 [55]	.873	.586	.701	.701	.725	.713	.581	.335	.425	.786	.197	.315	.565	.360	.440
CodeGPT [34]	.901	.575	.702	.721	.716	.719	.631	.456	.529	.789	.280	.413	.644	.660	.652
GraphCodeBERT [26]	.922	.573	.707	.741	.709	.725	.581	.407	.479	.785	.308	.442	.552	.341	.422
PLBART [5]	.837	.456	.590	.675	.696	.686	.616	.523	.566	.673	.284	.400	.411	.233	.297
CoTexT [45]	.928*	.586	.718	.583	.731	.649	.593	.545	.568	.764	.275	.404	.642	.560	.599

Table 4: Statistical test results of Cliff’s δ between all the hypergraph/graph neural networks on precision, recall, and f1-measure score.

Neural Network Pair	Precision	Recall	F1-Measure
HGNN vs HGNN+	1 (large)	-1 (large)	-0.333 (medium)
HGNN vs GraphSAGE	1 (large)	0.528 (large)	0.944 (large)
HGNN vs GCN	1 (large)	-1 (large)	0.667 (large)
HGNN vs GAT	1 (large)	0.278 (small)	0.833 (large)
HGNN+ vs GraphSAGE	0.778 (large)	1 (large)	1 (large)
HGNN+ vs GCN	1 (large)	-1 (large)	0.944 (large)
HGNN+ vs GAT	0.833 (large)	1 (large)	0.944 (large)
GraphSAGE vs GCN	1 (large)	-1 (large)	-1 (large)
GraphSAGE vs GAT	0.05 (negligible)	0.05 (negligible)	-0.056 (negligible)
GCN vs GAT	-0.667 (large)	1 (large)	0.611 (large)

Table 5: Performance of various refactoring tools on *ECAC-eval* [53].

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- α	60.1%	68.6%	64.1%
HECS- β	76.8%	84.3%	80.4%



Q1. Years of Experience in Software Development				
12%	22%	42%	18%	6%
Less than 1 year	1–3 years	4–5 years	6–10 years	More than 10 years
Q2. How often do you consider using Extract Class Refactoring in your projects?				
10%	20%	32%	22%	16%
Never	Rarely	Sometimes	Often	Always
Q3. Would you prefer to use tools for detecting Extract Class Refactoring opportunities?		Q4. Which of these tools do you recommend?		
32%	38%	30%	10%	18%
Yes	Maybe	No	Tool 1	Tool 2
			Tool 3	Tool 4

Note: Tool1 denote LLMRefactor, Tool 2 denote JDeodorant, Tool 3 denote SSECS and Tool 4 denote HECS

Figure 9. Questionnaire results achieved in our user study.

review suggestion results on *ECHumanEval* [32] for 4 refactoring tools used in RQ2 (14×4 results in total), including LLMRefactor, JDeodorant, SSECS, and HECS- β . To guarantee that participants are unaware of which tool we developed, we blindly display each group in a random order to each participant. After reviewing, participants are encouraged to evaluate the performance of each refactoring tool and answer Q1-Q4 in Table 7. Details of all the questionnaire results are available [3].

Fig. 9 presents the results of our survey. 42% of participants reported 4 to 5 years of experience in software development. 10% of participants never considered Extract Class refactoring during development. 30% of participants reported that Extract Class refactoring can be conducted without the aid of tools. After reviewing all the results of various refactoring tools, the results revealed that 64% of participants recommend HECS compared to other refactoring tools.

Table 6 presents the average accuracy of each refactoring tool on each instance of *ECHumanEval* [53]. We observed the HECS also outperforms other refactoring tools. We further analyze an illustrated example to demonstrate the difference between HECS and other refactoring tools in practice. Table 8 presents the suggested results of 4 refactoring tools including LLMRefactor, JDeodorant, SSECS, and HECS- β of target class: `IDDatatypeValidator` highlighted with “*” in Table 6. The column: Ground-Truth represents the applied refactoring results. We use  and  to highlight the truly and falsely extracted methods or fields, respectively. As presented in Table 8, we observed that LLMRefactor falsely reports 3 fields and 1 method. LLMRefactor also misses 1 field. JDeodorant falsely reports 1 field and 2 methods but misses 3 fields. SSECS falsely reports 1 method but misses 3 fields. In comparison, HECS- β merely misses 1 field. HECS can provide more comprehensive and precise results than other refactoring tools: LLMRefactor, JDeodorant, and SSECS.

Answer to RQ3: HECS is more practical compared to other state-of-the-art Extract Class refactoring tools for 64% of participants.

5 Discussion

In this section, we discuss the threats to validity, limitations, and applications of our approach.

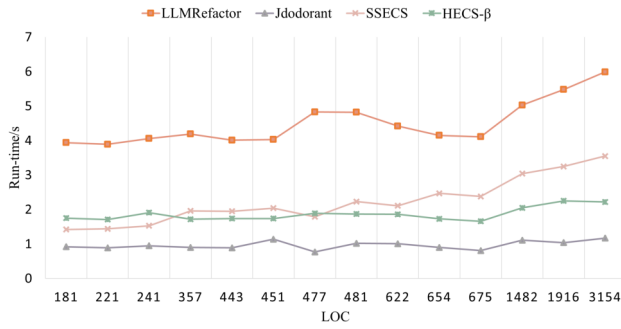
Threats to Validity. The first threat is the quality of our training data. Noise may exist and cause the bias of model training. We will further keep reviewing the training data and filtering the noise. The second threat exists in the pre-condition verification. On average, almost 20% of candidates are filtered. We heuristically employ several pre-conditions with large language model and further explore more pre-conditions. The third threat comes from the run-time overhead of HECS. Figure 10 presents the run-time overhead of HECS on *ECHumanEval* [32]. Given a target class, we observed that HECS- β responses within 2s on average. In comparison, LLMRefactor takes approximately 2 times longer than HECS. JDeodorant

Table 6: Accuracy of various refactoring tools on *ECHumanEval* [32]. P: Precision, R: Recall, F1: F1-Measure.

Subject	Target Class	LOC	#EClass	#EMethod	#EField	LLMRefactor			JDeodorant			SSECS			HECS- β		
						P	R	F1	P	R	F1	P	R	F1	P	R	F1
Xerces	DOMParser	3154	3	4	4	.059	.333	.1	.063	.333	.106	.235	.167	.195	.56 ⁺	.979 ⁺	.712 ⁺
	IDDatatypeValidator*	181	1	1	4	.5	.8	.615	.4	.4	.4	.5	.2	.286	1 ⁺	.8 ⁺	.889 ⁺
	NodeImpl	1220	1	1	2	.04	.333	.071	.188	1 ⁺	.316	.393	.333	.361	.645 ⁺	.606	.625 ⁺
	Serializer	221	1	4	0	.667	.5	.572	.45	.6	.514	1 ⁺	.5	.667	.615	1 ⁺	.762 ⁺
	XMLDTDValidator	1916	1	0	13	.325	1 ⁺	.491	.148	1 ⁺	.258	.128	.077	.096	1 ⁺	.8	.889 ⁺
	XMLSerializer	1482	1	12	1	.292	.5	.369	.462	.429	.445	.594	.679 ⁺	.634 ⁺	.667 ⁺	.5	.572
	XSParticleDecl	241	2	3	2	.105	1 ⁺	.19	.143	.643	.234	.5	.25	.333	1 ⁺	.875	.933 ⁺
GanttProject	ChartModelBase	481	2	5	6	.046	.75	.087	.286	1 ⁺	.445	.557 ⁺	.975	.709	.556	1 ⁺	.715 ⁺
	ChartModelImpl	470	1	2	0	.059	1 ⁺	.111	.333	1 ⁺	.5	.455	1 ⁺	.625	1 ⁺	.933	.965 ⁺
	DependencyGraph	675	1	2	4	.06	1 ⁺	.113	.111	.333	.167	.405	.711	.516	.462 ⁺	.956	.623 ⁺
	ExporterToHTML	357	1	6	0	.2	.667	.308	.667	.625	.645	.516	.889	.653	.727 ⁺	.889 ⁺	.8 ⁺
	GPViewImpl	443	1	1	2	.182	.667	.286	.25	.333	.286	.333	.333	.333	1 ⁺	.667 ⁺	.77 ⁺
	GanttTaskPropertiesBean	477	2	4	1	.17	.889	.285	.22	1 ⁺	.361	.342	.417	.376	.693 ⁺	.867	.8 ⁺
	GanttResourcePanel	653	1	21	33	.063	1 ⁺	.119	.188	1 ⁺	.316	.438	1 ⁺	.609	.83 ⁺	.929	.877 ⁺

Table 7: Extract Class refactoring tools questionnaire.

Participants Information
Q1. Years of Experience in Software Development <input type="checkbox"/> Less than 1 year <input type="checkbox"/> 1 – 3 years <input type="checkbox"/> 4 – 5 years <input type="checkbox"/> 6 – 10 years <input type="checkbox"/> More than 10 years
Tools Evaluation
Q2. How often do you consider using Extract Class Refactoring in your projects? <input type="checkbox"/> Never <input type="checkbox"/> Rarely <input type="checkbox"/> Sometimes <input type="checkbox"/> Often <input type="checkbox"/> Always
Q3. Would you prefer to use tools for detecting Extract Class Refactoring opportunities? <input type="checkbox"/> Yes <input type="checkbox"/> Maybe <input type="checkbox"/> No
Q4. Which of these tools do you recommend? (Please select only one) <input type="checkbox"/> Tool 1 <input type="checkbox"/> Tool 2 <input type="checkbox"/> Tool 3 <input type="checkbox"/> Tool 4
Additional Comments
Do you have any suggestions for features or improvements in extract class refactoring tools? Open Question

**Figure 10. Run-time overhead of HECS on *ECHumanEval* [32].**

responded within 1s, which is less than 50% of HECS’s run-time. SSECS exhibited a close response time compared with HECS- β . We will further improve HECS’s run-time performance by parallelizing and vectorizing computations.

Limitations. The first limitation comes from hyper-parameters. We followed the default hyper-parameters in previous literature

and will further investigate the impact of hyper-parameters on the performance. Second, our tool relies on existing implementations of static analysis tool and hypergraph neural network, which may contain bugs and affect the performance. We will continue to report found bugs and integrate the latest patches into our tool. Third, we only focus on Java, and will further explore more programming languages. Fourth, We have not fully explored the model interpretability and further investigate more interpretable approaches. Fourth, HECS aims to identify possible extraction solutions, which can serve as input for IDEs to assist developers in splitting large classes. In current version, HECS does not fully support the automated application of these refactoring opportunities, such as updating invocations and building interconnections of extracted classes, which can further be studied and improved.

Applications. Our research can be further extended in several directions: First, our results demonstrate that hypergraph neural network is practical and our approach may also be used for other extract-related refactoring approaches like Extract Variable and Extract Method. Second, we currently employ GPT-3.5 model to implement pre-condition analysis, and will keep exploring more large language models. Our results also motivate us to further build a unified large language model for extract-related refactoring tasks.

6 Related Work

We introduce related techniques that motivate our work as follows:

Extract Class Refactoring Approaches. Opdyke et al. [41] first designed the Extract Class refactoring algorithm by exploiting aggregations and reusable components. Gabriele et al. [13] introduced a weighted graph based on Jaccard distance and subgraph segmentation to detect Extract Class refactoring opportunities. Akash et al. [7] introduced the semantic dependence between methods by calculating the cosine similarity among topic distributions for each method, which largely improves the performance of Extract Class refactoring opportunity detection. Zhao et al. [60] introduced an innovative method, named Butterfly Space, leveraging static and dynamic method-level dependencies to identify Extract Class refactoring opportunities. Jeba et al. [30] leveraged context metric and hierarchical clustering in suggesting Extract Class refactoring

Table 8: Detected refactoring candidates of extracted class: IDDatatypeValidator with various refactoring tools.

Label	Field or Method Declaration	Ground-Truth	LLMRefactor	JDdeodorant	SSECS	HECS- β
Field1	private Object fNullValue = null;		✓			
Field2	private DatatypeMessageProvider fMessageProvider = new DatatypeMessageProvider();		✓	✓		
Field3	private Hashtable fTableOfId;					
Field4	private Locale fLocale = null;	✓	✓	✓		✓
Field5	private int fFacetsDefined = 0;	✓	✓		✓	✓
Field6	private int fLength = 0;	✓	✓			✓
Field7	private Vector fEnumeration = new Vector();	✓				
Field8	public static final int IDREF_STORE = 0;		✓			
Field9	public static final int ID_CLEAR = 1;					
Method1	public IDDatatypeValidator() {...}					
Method2	public Object validate(String content, Object IDStorage) {...}					
Method3	public Object clone() {...}			✓	✓	
Method4	private boolean addId(String content, Object idTable) {...}			✓		
Method5	public void setLocale(Locale locale) {...}	✓	✓	✓		✓
Method6	private String getErrorString(int major, int minor, Object args[]) {...}		✓			

to improve accuracy. Akash [6] explored Variational Graph Auto-Encoders (VGAE) to derive features and used clustering to suggest Extract Class refactoring.

Machine learning-based Refactoring Approaches. Xu et al. [57] developed a machine learning-based approach to identify Extract Method refactoring opportunities, which encodes several metrics as features including complexity, cohesion, and coupling. Mikolov et al. [37] utilized the word2vec technique to generate implicit representations and train a Convolutional Neural Network (CNN) model for detecting code smells. Hadj-Kacem et al. [27] extracted implicit representations from abstract syntax trees with Variational Auto Encoder (VAE) [43] and fed them into the logistic regression classifier to detect code smells. Cui et al. [17] employed graph embedding techniques to automatically learn features from code and train classifiers to guide Move Method refactoring.

Compared to previous work, our approach first introduces the concept of hypergraph and uses hypergraph neural network for suggesting Extract Class refactoring candidates.

7 Conclusion

In this paper, we proposed HECS, a novel hypergraph-learning method for Extract Class refactoring suggestions by leveraging historical samples. Our approach starts with extracting intra-class dependency graphs, converting them into intra-class dependency hypergraphs, and utilizing an enhanced hypergraph neural network alongside a large language model (LLM) to develop a refactoring suggestion system. The findings demonstrate HECS showed a 38.5% increase in precision, a 9.7% increase in recall, and a 44.4% increase in f1-measure over leading tools like JDdeodorant, SSECS, and LLMRefactor, with 64% of participants finding it more practical for real-world applications. Specifically, hypergraph neural networks were found to be more effective than graph neural networks. Overall, HECS advances Extract Class refactoring by integrating hypergraph neural networks with large language models, marking a significant step forward in software engineering tools.

Data-Availability Statement

The data and software used in this study, including the Hypergraph learning-based Extract Class refactoring detection System (HECS),

are publicly available. The associated artifact can be accessed on Zenodo [2]. This repository contains all the necessary materials to replicate the experiments and validate the findings reported in this paper.

Acknowledgement

We would like to thank anonymous reviewers for their insightful and constructive feedback. This work was supported by National Natural Science Foundation of China (62202357, U21B2015), Proof of Concept Foundation of Xidian University Hangzhou Institute of Technology under Grant (XJ2023230039), Natural Science Foundation of Jiangsu Province (BK202302028), Open Research Fund of Shanghai Key Laboratory of Trustworthy Computing (East China Normal University).

References

- [1] [n. d.]. GanttProject. <https://sourceforge.net/projects/ganttproject/>. Accessed: July 3, 2023.
- [2] 2024. *Artifact Link*. <https://doi.org/10.5281/zenodo.12662219>
- [3] 2024. *Data Link*. <https://github.com/cnzn/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] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
- [6] Pritom Saha Akash and Kevin Chen-Chuan Chang. 2022. Exploring Variational Graph Auto-Encoders for Extract Class Refactoring Recommendation. *arXiv:arXiv:2203.08787*
- [7] 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*.
- [8] Lulwah Alkulaib and Chang-Tien Lu. 2023. Balancing the Scales: HyperSMOTE for Enhanced Hypergraph Classification. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 5140–5145.
- [9] Musaad Alzahrani. 2022. Extract Class Refactoring Based on Cohesion and Coupling: A Greedy Approach. *Computers* 11, 8 (2022). <https://doi.org/10.3390/computers11080123>
- [10] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.
- [11] 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.
- [12] 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.

- [13] 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.
- [14] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Andrian Marcus, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2014. In medio stat virtus: Extract class refactoring through nash equilibria. In *Software Maintenance, Reengineering & Reverse Engineering*.
- [15] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.
- [16] 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.
- [17] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMov: 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.
- [18] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Systems with Applications* 249 (2024), 123602.
- [19] 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.
- [20] Elena Litani. [n. d.]. Commit id of motivated example. <https://github.com/apache/xerces2-j/commit/dc38ef3f474431c6ee975d071441496ac221a110/>. Accessed: Apr 10, 2001.
- [21] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 3558–3565.
- [22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- [23] Martin Fowler. 1997. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- [24] Yue Gao, Yifan Feng, Shuyi Ji, and Rongrong Ji. 2022. HGNN+: General hypergraph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 3 (2022), 3181–3199.
- [25] Google BigQuery. [n. d.]. Google BigQuery. <https://console.cloud.google.com/marketplace/details/github/github-repos>. Accessed: 2021.
- [26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, and S. Fu. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.
- [27] 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.
- [28] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
- [29] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- [30] 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).
- [31] 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.
- [32] 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>
- [33] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
- [34] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codeglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
- [35] Tom Mens, Gabriele Taentzer, and Olga Runge. 2007. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling* 6, 3 (2007).
- [36] 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.
- [37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
- [38] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.
- [39] 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.
- [40] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.
- [41] W. F. Opdyke and R. E. Johnson. 1992. Refactoring Object-Oriented Frameworks. *University of Illinois at Urbana-Champaign* (1992).
- [42] OpenAI. [n. d.]. Introducing ChatGPT. <https://openai.com/blog/chatgpt/>. Accessed: July 3, 2023.
- [43] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.
- [44] 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.
- [45] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645* (2021).
- [46] Yu Qu and Heng Yin. 2021. Evaluating network embedding techniques' performances in software bug prediction. *Empirical Software Engineering* 26, 4 (2021), 1–44.
- [47] 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.
- [48] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanabe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples. *arXiv preprint arXiv:2311.11690* (2023).
- [49] 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.
- [50] F. Simon, F. Steinbrückner, and C. Lewerentz. 2001. Metrics based refactoring. In *European Conference on Software Maintenance & Reengineering*.
- [51] The Apache Software Foundation. [n. d.]. Xerces. <https://xerces.apache.org/>. Accessed: July 3, 2023.
- [52] 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.
- [53] N. Tsantalis, A. S. Ketkar, and D. Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* PP, 99 (2020), 1–1.
- [54] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.
- [55] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [56] 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.
- [57] 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.
- [58] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.
- [59] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. 2021. Graphsmote: Imbalanced node classification on graphs with graph neural networks. In *Proceedings of the 14th ACM international conference on web search and data mining*. 833–841.
- [60] 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 2024-04-12; accepted 2024-07-03