Three Heads Are Better Than One: Suggesting Move Method Refactoring Opportunities with Inter-class Code Entity Dependency Enhanced Hybrid Hypergraph Neural Network

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

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

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

Peng Ji
Xidian University
Xi'an, China
jipeng@stu.xidian.edu.cn

Minglang Qiao
Xidian University
Xi'an, China
qiaoml@stu.xidian.edu.cn

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

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

Methods implemented in incorrect classes will cause excessive reliance on other classes than their own, known as a typical code smell symptom: feature envy, which makes it difficult to maintain increased coupling between classes. Addressing this issue, several Move Method refactoring tools have been proposed, employing a two-phase process: identifying misplaced methods to move and appropriate classes to receive, and implementing the mechanics of refactoring. These tools traditionally use hard-coded metrics to measure correlations between movable methods and target classes and apply heuristic thresholds or trained classifiers to unearth refactoring opportunities. Yet, these approaches predominantly illuminate pairwise correlations between methods and classes while overlooking the complex and complicated dependencies binding multiple code entities within these methods/classes that are prevalent in real-world cases. This narrow focus can lead to refactoring suggestions that may diverge from developers' actual needs. To bridge this gap, our paper leverages the concept of inter-class code entity dependency hypergraph to model complicated dependency relationships involving multiple code entities within various methods/classes and proposes a hypergraph learning-based approach to suggest Move Method refactoring opportunities named HMove. We first construct inter-class code entity dependency hypergraphs from training samples and assign attributes to entities with a pre-trained code model. All the attributed hypergraphs are fed into a hybrid hypergraph neural network for training. Utilizing this trained neural network alongside a large language model, we construct a refactoring suggestion system. We trained HMove on a large-scale dataset and evaluated it on two real-world datasets. The results show that demonstrates an increase of 27.8% in precision, 2.5% in recall, and 18.5% in f1-measure compared to 9 state-of-the-art refactoring tools, which is more useful for 68% of participants. The results also unveil practical suggestions and new insights that benefit existing feature envy-related refactoring techniques.

Qingshan Li is the corresponding author.

CCS CONCEPTS

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

KEYWORDS

Move Method Refactoring, Hypergraph Neural Network

ACM Reference Format:
Di Cui, Jiaqi Wang, Qiangqiang Wang, Peng Ji, Minglang Qiao, Yutong Zhao, Jingzhao Hu, Luqiao Wang, and Qingshan Li. 2024. Three Heads Are Better Than One: Suggesting Move Method Refactoring Opportunities with Inter-class Code Entity Dependency Enhanced Hybrid Hypergraph Neural Network. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695068

1 INTRODUCTION

The placement of methods within classes is usually guided by conceptual criteria and rigorous metrics. However, during software evolution, developers may inadvertently implement methods in improper classes, causing excessive reliance on other classes rather than its own, known as a typical code smell symptom: feature envy. To mitigate this issue, the Move Method refactoring is applied to relocate the misplaced method from its current class to the appropriate one with a clearer distribution of responsibilities among classes.

Move Method refactoring involves two phases: (i) suggesting misplaced methods to move and appropriate classes to receive, and (ii) applying mechanics to perform refactoring, which improves code's internal structure without altering external behaviors [48].

Several automatic Move Method refactoring tools have been proposed [16, 31–33, 36, 46–48, 56]. Most of these tools first calculate hard-coded metrics to measure correlations between movable methods and target classes. They further apply heuristic thresholds or train classifiers based on these metrics to suggest refactoring opportunities. However, in most cases, these tools merely focus on pairwise correlations between methods and classes and fail to capture the complex dependencies binding multiple code entities within these methods/classes, which can be challenging to represent rich movement cases in practice. This may result in suggested refactoring opportunities that do not align with developers' preferences.

To more comprehensively capture the relationships among multiple code entities and enhance the suggestion accuracy, we introduce the concept of inter-class code entity dependency hypergraph. Beyond pairwise edges, this advanced representation allows for edges that bind multiple nodes, reflecting the reality that multiple code entities often interact in unison. The hypergraph paradigm, with its ability to embody a cohesive group of code entities, sets the stage for analyzing code entity dependency more precisely and suggesting refactoring opportunities more efficiently.

The workflow of our HMove approach is as follows: We begin by extracting code entities and their dependencies from both training and testing samples, forming inter-class code entity dependency hypergraphs. Subsequently, the pre-trained code model generates vector embeddings for entities within the hypergraphs, which are then used to train the hybrid hypergraph neural network. Finally, we employ the trained neural network and a large language model (LLM) to develop a refactoring suggestion system that identifies and relocates misplaced methods into appropriate classes. HMove was trained on a large-scale dataset and evaluated against two real-world datasets, with systematic comparisons to nine state-of-the-art tools: LLMRefactor [46], FeTruth [32], FeDeep [33], PathMove [31], JDeodorant [48], JMove [47], RMove [16], FeGNN [56], and FePM [36]. The results suggest that our HMove approach surpasses these state-of-the-art tools in both effectiveness and usefulness. Additionally, the results reveal insightful and practical implications of hypergraph utilization, benefiting other feature envy-related refactoring approaches such as Move Field refactoring and Extract+Move Method refactoring.

In summary, we make the following contributions:

- A new perspective of hypergraph to suggest Move Method refactoring opportunities.
- A systematic exploration of implementations of HMove on combinations of 6 pre-trained code models, 3 graph neural networks, and 3 hypergraph neural networks. The experimental results reveal that the following combinations achieve the best results: 1) GCN+HGNN+CodeT5, 2) GraphSAGE+HGNN+CodeT5, and 3) GraphSAGE+HGNN$^+$+CodeT5.
- A comprehensive evaluation of HMove on two real-world datasets. HMove demonstrates an increase of 27.8% in precision, 2.5% in recall, and 18.5% in F1-measure compared with the best results of 9 state-of-the-art tools, including LLMRefactor [46], FeTruth [32], FeDeep [33], PathMove [31], JDeodorant [48], JMove [47], RMove [16], FeGNN [56], and FePM [36]. The questionnaire results also indicate that HMove is more useful than state-of-the-art tools for 68% of participants.
- A benchmark to investigate the effectiveness of hypergraph neural networks in suggesting Move Method refactoring opportunities. All the data is publicly available [2].

2 PRELIMINARY

2.1 Problem Definition

In this section, we aim to furnish precise definitions for the terminologies employed in our paper.

Move Method Refactoring Suggestion. The Move Method refactoring suggestion can be regarded as discovering a set of movable methods and target classes from the source code, which is defined as *MoveMethodSet*. Each item in *MoveMethodSet* can be modeled as a set of three elements, which is formally defined as follows:

$$MoveMethodSet = \{(m_i, sc_i, tc_i) \mid i = 1, 2, \cdots, k\} \tag{1}$$

where m_i represents the potentially movable method. sc_i represents the source class to which the method: m_i belongs. tc_i represents the corresponding target class for the method: m_i.

Code Entity and Code Entity Dependency. A code entity is an object with a given name or identifier from the source code. Code entity dependency describes relationships between code entities. There is a code entity dependency from entity: e_i to entity: e_j when entity: e_i depends on entity: e_j to complete its functionality. In our paper, we focus on 2 types of code entities including method and field, and 3 types of code entity dependencies including method call, field access, and joint class contain (two entities within the same class will be linked by a common hyperedge, creating a class-wide dependency that connects all entities in that class).

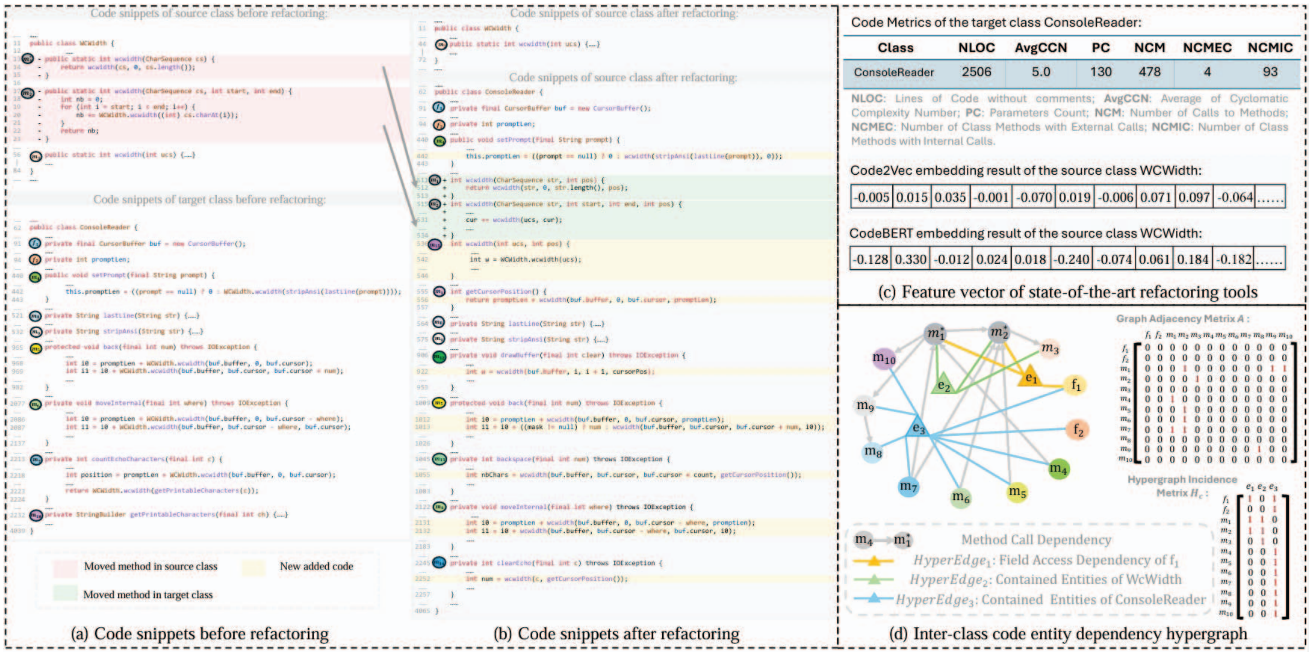
Inter-class Code Entity Dependency Hypergraph. For a pair of source class and target class: (sc, tc), the Inter-class Entity Dependency Hypergraph, denoted as $HG_c = (V_c, E_c, \mathcal{E}_c, \mathcal{W}_c)$, is defined as follows: V_c represents the code entities of field and method within the class pair (sc, tc), while E_c denotes the method-call entity dependencies within the node set V_c, represented as an adjacency matrix of a directed graph. Additionally, the set of field-access and joint-class-contain entity dependencies are defined in $\mathcal{E}_c$, with a diagonal matrix $\mathcal{W}_c$ representing the weights for each hyperedge. Each hyperedge is assigned a weight of 1.0 by default. $\mathcal{E}_c$ represents a collection of hyperedges, delineated by a hypergraph incidence matrix, wherein each hyperedge consists of a set of vertices linked to variable access or parameter passing relationships with the target method being analyzed. Thus, hypergraph incidence matrix $\mathbf{H}_c$, where entries can be defined as follows:

$$h(v, e) = \begin{cases} 1 & \text{if } v \in e, \\ 0 & \text{if } v \notin e. \end{cases} \tag{2}$$

For a hyperedge $e \in \mathcal{E}_c$, its edge degree is defined as $\delta(e) = \sum_{v \in \mathcal{V}_c} \mathbf{H}_c(v, e)$.

2.2 Motivation Example

We present a real-world example of commit 1eb3b62 [1] in an open-source project: jline2 [3] to illustrate the Move Method Refactoring



Code snippets of source class before refactoring:

```
11   public class WCWidth {
12       ......
13   - public static int wcwidth(CharSequence cs) {
14   -     return wcwidth(cs, 0, cs.length());
15   - }
16
17   - public static int wcwidth(CharSequence cs, int start, int end) {
18   -     int nb = 0;
19   -     for (int i = start; i < end; i++) {
20   -         nb += WCWidth.wcwidth((int) cs.charAt(i));
21   -     }
22   -     return nb;
23   - }
56     public static int wcwidth(int ucs) {.....}
       .....
84   }
```

Code snippets of target class before refactoring:

```
62   public class ConsoleReader {
       .....
91     private final CursorBuffer buf = new CursorBuffer();
       .....
94     private int promptLen;
       .....
440    public void setPrompt(final String prompt) {
           .....
442        this.promptLen = ((prompt == null) ? 0 : WCWidth.wcwidth(stripAnsi(lastLine(prompt))));
443    }
       .....
521    private String lastLine(String str) {.....}
       .....
532    private String stripAnsi(String str) {.....}
       .....
965    protected void back(final int num) throws IOException {
           .....
968        int i0 = promptLen + WCWidth.wcwidth(buf.buffer, 0, buf.cursor);
969        int i1 = i0 + WCWidth.wcwidth(buf.buffer, buf.cursor, buf.cursor + num);
           .....
982    }
       .....
2077   private void moveInternal(final int where) throws IOException {
           .....
2086       int i0 = promptLen + WCWidth.wcwidth(buf.buffer, 0, buf.cursor - where);
2087       int i1 = i0 + WCWidth.wcwidth(buf.buffer, buf.cursor - where, buf.cursor);
           .....
2137   }
       .....
2213   private int countEchoCharacters(final int c) {
           .....
2218       int position = promptLen + WCWidth.wcwidth(buf.buffer, 0, buf.cursor);
           .....
2223       return WCWidth.wcwidth(getPrintableCharacters(c));
2224   }
       .....
2232   private StringBuilder getPrintableCharacters(final int ch) {.....}
       .....
4039 }
```

🟥 Moved method in source class	🟨 New added code
🟩 Moved method in target class	

(a) Code snippets before refactoring

Code snippets of source class after refactoring:

```
11   public class WCWidth {
       .....
44     public static int wcwidth(int ucs) {.....}
       .....
72   }
```

Code snippets of source class after refactoring:

```
62   public class ConsoleReader {
       .....
91     private final CursorBuffer buf = new CursorBuffer();
       .....
94     private int promptLen;
       .....
440    public void setPrompt(final String prompt) {
           .....
442        this.promptLen = ((prompt == null) ? 0 : wcwidth(stripAnsi(lastLine(prompt)), 0));
443    }
       .....
511  + int wcwidth(CharSequence str, int pos) {
512  +     return wcwidth(str, 0, str.length(), pos);
513  + }
515  + int wcwidth(CharSequence str, int start, int end, int pos) {
     +     .....
531  +     cur += wcwidth(ucs, cur);
     +     .....
534  + }
536    int wcwidth(int ucs, int pos) {
           .....
542        int w = WCWidth.wcwidth(ucs);
           .....
544    }
       .....
555    int getCursorPosition() {
556        return promptLen + wcwidth(buf.buffer, 0, buf.cursor, promptLen);
557    }
       .....
564    private String lastLine(String str) {.....}
       .....
575    private String stripAnsi(String str) {.....}
       .....
906    private void drawBuffer(final int clear) throws IOException {
           .....
922        int w = wcwidth(buf.buffer, i, i + 1, cursorPos);
           .....
953    }
       .....
1009   protected void back(final int num) throws IOException {
           .....
1012       int i0 = promptLen + wcwidth(buf.buffer, 0, buf.cursor, promptLen);
1013       int i1 = i0 + ((mask != null) ? num : wcwidth(buf.buffer, buf.cursor, buf.cursor + num, i0));
           .....
1026   }
       .....
1045   private int backspace(final int num) throws IOException {
           .....
1055       int nbChars = wcwidth(buf.buffer, buf.cursor, buf.cursor + count, getCursorPosition());
           .....
1083   }
       .....
2122   private void moveInternal(final int where) throws IOException {
           .....
2131       int i0 = promptLen + wcwidth(buf.buffer, 0, buf.cursor - where, promptLen);
2132       int i1 = i0 + wcwidth(buf.buffer, buf.cursor - where, buf.cursor, i0);
           .....
2183   }
       .....
2245   private int clearEcho(final int c) throws IOException {
           .....
2252       int num = wcwidth(c, getCursorPosition());
           .....
2257   }
       .....
4065 }
```

(b) Code snippets after refactoring

Code Metrics of the target class ConsoleReader:

Class	NLOC	AvgCCN	PC	NCM	NCMEC	NCMIC
ConsoleReader	2506	5.0	130	478	4	93

NLOC: Lines of Code without comments; **AvgCCN**: Average of Cyclomatic Complexity Number; **PC**: Parameters Count; **NCM**: Number of Calls to Methods; **NCMEC**: Number of Class Methods with External Calls; **NCMIC**: Number of Class Methods with Internal Calls.

Code2Vec embedding result of the source class WCWidth:

-0.005	0.015	0.035	-0.001	-0.070	0.019	-0.006	0.071	0.097	-0.064	......

CodeBERT embedding result of the source class WCWidth:

-0.128	0.330	-0.012	0.024	0.018	-0.240	-0.074	0.061	0.184	-0.182	......

(c) Feature vector of state-of-the-art refactoring tools

Graph Adjacency Metrix A :

$$\begin{array}{c|cccccccccccc} & f_1 & f_2 & m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 & m_9 & m_{10} \\ \hline f_1 & 0&0&0&0&0&0&0&0&0&0&0&0 \\ f_2 & 0&0&0&0&0&0&0&0&0&0&0&0 \\ m_1 & 0&0&0&1&0&0&0&0&0&0&1&1 \\ m_2 & 0&0&0&0&1&0&0&0&0&0&0&0 \\ m_3 & 0&0&0&0&0&0&0&0&0&0&0&0 \\ m_4 & 0&0&1&0&0&0&0&0&0&0&0&0 \\ m_5 & 0&0&0&1&0&0&0&0&0&0&0&0 \\ m_6 & 0&0&0&1&0&0&0&0&0&0&0&0 \\ m_7 & 0&0&1&1&0&0&0&0&0&0&0&0 \\ m_8 & 0&0&0&0&0&0&0&0&0&0&0&0 \\ m_9 & 0&0&0&0&0&0&0&0&0&1&0&0 \\ m_{10} & 0&0&0&0&0&0&0&0&0&0&0&0 \end{array}$$

Hypergraph Incidence Metrix H_c :

$$\begin{array}{c|ccc} & e_1 & e_2 & e_3 \\ \hline f_1 & 1&0&1 \\ f_2 & 0&0&1 \\ m_1 & 1&1&0 \\ m_2 & 1&1&0 \\ m_3 & 0&1&0 \\ m_4 & 0&0&1 \\ m_5 & 0&0&1 \\ m_6 & 0&0&1 \\ m_7 & 0&0&1 \\ m_8 & 0&0&1 \\ m_9 & 0&0&1 \\ m_{10} & 0&0&1 \end{array}$$

m_1^*
m_2^*
m_3
m_{10}
e_2
e_1
f_1
m_9
e_3
f_2
m_8
m_4
m_7
m_6
m_5
$m_4 \rightarrow m_1^*$ Method Call Dependency
$HyperEdge_1$: Field Access Dependency of f_1
$HyperEdge_2$: Contained Entities of WcWidth
$HyperEdge_3$: Contained Entities of ConsoleReader

(d) Inter-class code entity dependency hypergraph

Figure 1. A motivated example of Move Method refactoring operation and inter-class code entity dependency hypergraph.

operation, which is demonstrated in Fig. 1. The two classes, including WCWidth and ConsoleReader, in the jline2 project exhibit the Feature Envy code smell and have a complex interdependency, making them prime candidates for the Move Method refactoring recommendation.

Fig. 1.(a) shows code snippets from the source class: WCWidth, containing three code entities: m_1^*, m_2^*, and m_3 and the target class: ConsoleReader with 9 code entities: f_1, f_2, m_4, m_8, m_9, m_5, m_6, m_7, and m_{10} before the Move Method refactoring, where m_1^* and m_2^* are methods that should be moved to the target class. Lines 13 to 15 and 17 to 23 highlighted with red colour and "-" marks, can further be moved from the source class: WCWidth to the target class: ConsoleReader for these two methods: m_1^* and m_2^* are frequently invoked by the fields/methods of the target class: ConsoleReader. Fig. 1.(b) presents the corresponding code snippets of the source class: WCWidth and the target class: ConsoleReader after Move Method refactoring. In Fig. 1.(b), we highlight moved methods with green colour and "+" marks. Line 511 to 513 and 515 to 534 map two moved methods respectively. Line 442, 536 to 544, 556, 922, 1012, 1013, 1055, 2131, 2132 and 2252 with light yellow colour further update the invocation of these two moved methods.

For the code snippet in Fig. 1.(a), Fig. 1.(c) and Fig. 1.(d) present the code representation of WCWidth with three typical feature vectors generated by state-of-the-art refactoring tools and hyperedge of inter-class code entity dependency hypergraph respectively (construction detail will be illustrated in Section III). We observed that CodeBERT [21] and Code2Vec [7] offer superior code snippet representation, with increased vector length and variance, compared to traditional code metrics [56]. However, these representations fail to precisely capture code entity dependency, which is proven to have a significant impact on Move Method refactoring suggestion task in previous work [16, 56]. Intuitively, CodeBERT's performance can be enhanced when introducing inter-class code entity dependency hypergraph. Thus in our paper, we will fully exploit the combination of inter-class code entity dependency hypergraph and pre-trained code model, and propose a hypergraph learning-based approach: HMove for suggesting Move Method refactoring opportunities.

3 METHODOLOGY

Fig. 2 presents an overview of the HMove pipeline that aims to suggest Move Method refactoring candidates automatically. HMove is structured into two phases: training and detection. The training phase includes hypergraph construction (Section 3.1), entity attribute generation (Section 3.2), and hypergraph learning (Section 3.3), culminating in a well-trained model adept at identifying refactoring opportunities. During the detection phase (Section 3.4), attributed hypergraphs of projects are constructed and entity attributes are generated. These hypergraphs are then input into the trained model to produce refactoring candidates, which are subsequently refined through pre-condition verification using a large language model (LLM).

3.1 Hypergraph Construction

Given a subject as input, we first extract code entity and code entity dependency using *ENRE* [29], a state-of-the-art static analysis tool for extraction of code entity dependencies. *ENRE* supports more than 11 types of code entity dependencies and provides flexible interfaces to implement the customized analyzer. For each type of code entity dependency, we analyze and construct hyperedge respectively. Specially, method call dependencies are represented as directed edges to indicate the method invocation process. Merging method call dependencies into hyperedges may obscure this process, leading to a loss of information about the call sequence. We devise a heuristic algorithm to construct the inter-class code entity dependency hypergraph depicted in Alg. 1. The input is code entity:

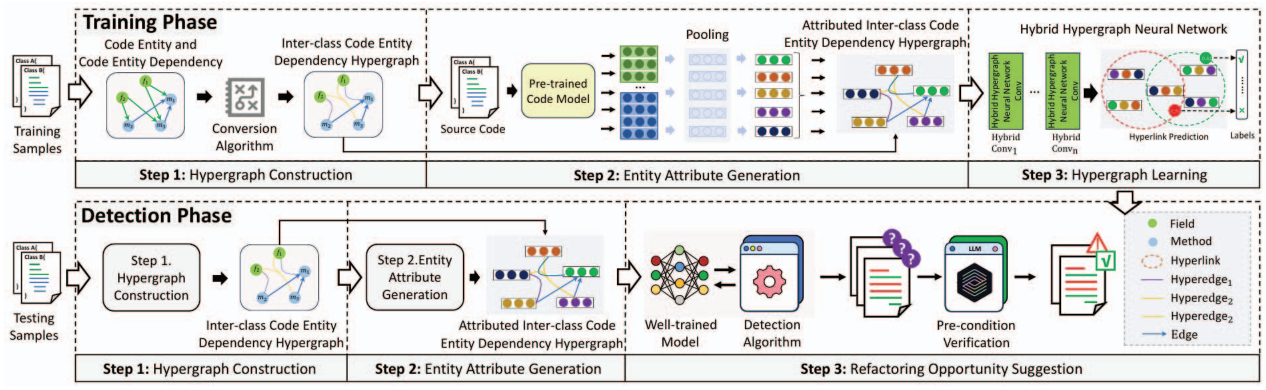

Training Phase
Code Entity and Code Entity Dependency
Conversion Algorithm
Inter-class Code Entity Dependency Hypergraph
Training Samples
Source Code
Pre-trained Code Model
Pooling
Attributed Inter-class Code Entity Dependency Hypergraph
Hybrid Hypergraph Neural Network
Hybrid Hypergraph Neural Network Conv
Hybrid Conv1
Hybrid Convn
Hyperlink Prediction
Labels
Step 1: Hypergraph Construction
Step 2: Entity Attribute Generation
Step 3: Hypergraph Learning
Detection Phase
Testing Samples
Step 1. Hypergraph Construction
Inter-class Code Entity Dependency Hypergraph
Step 2.Entity Attribute Generation
Attributed Inter-class Code Entity Dependency Hypergraph
Well-trained Model
Detection Algorithm
LLM
Pre-condition Verification
Field
Method
Hyperlink
Hyperedge1
Hyperedge2
Hyperedge2
Edge
Step 1: Hypergraph Construction
Step 2: Entity Attribute Generation
Step 3: Refactoring Opportunity Suggestion

Figure 2. Overview of the proposed approach: HMove including training phase and detection phase.

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

Model	Size	Architecture	Dataset
CodeBERT [21]	125M	RoBERTa (encoder)	CodeSearchNet [28]
GraphCodeBERT [25]	125M	RoBERTa (encoder)	CodeSearchNet [28]
CodeGPT [35]	124M	GPT2 (decoder)	CodeSearchNet [28]
CodeTrans [18]	220M	T5 (encoder-decoder)	CodeSearchNet [28] and Other [13, 38, 43, 55]
CodeT5 [52]	220M	T5 (encoder-decoder)	CodeSearchNet [28] and BigQuery [24]
CodeT5+ [51]	220M	T5 (encoder-decoder)	CodeSearchNet [28] and BigQuery [24]

V_e and code entity dependency: E_d. The output yields hypergraph: HG_c. Line 1 initializes the node set: V_c, edge set: E_c, and hyperedge set: $\mathcal{E}_c$ respectively. Line 2 to 12 iteratively examine each code entity dependency. Line 3 to 5 append the method-call dependency to E_c directly. Line 6 to 11 merge field-access dependencies with the same field as a hyperedge and append into $\mathcal{E}_c$. Line 13 to 17 iteratively examine each code entity and gather its joint containment relationship with the source class into $\mathcal{E}_c$. Line 19 finally assembles and returns the well-constructed hypergraph: HG_c.

Algorithm 1 Inter-class Code Entity Dependency Hypergraph Construction Algorithm

```
Input: $V_e, E_d$ - Code Entity and Code Entity Dependency
Output: $HG_c = (V_c, E_c, \mathcal{E}_c, \mathcal{W}_c)$ - Inter-class Code Entity Dependency
    Hypergraph
 1: $V_c, E_c, \mathcal{E}_c \leftarrow V_e, \emptyset, \emptyset$                         ▷ Initialization
 2: for all $e$ in $E_d$ do
 3:     if type$(e)$ == $METHOD\_CALL$ then                      ▷ construct edge
 4:         $E_c$.addEdge$(e)$
 5:     end if
 6:     if type$(e)$ == $FIELD\_ACCESS$ then              ▷ construct hyperedge
 7:         if $\mathcal{E}_c$.containHyperEdge$(e.FIELD)$ == $FALSE$ then
 8:             $\mathcal{E}_c$.addHyperEdge$((e.FIELD, \emptyset))$
 9:         end if
10:         $\mathcal{E}_c$.getHyperEdge$(e.FIELD)$.append$(e)$
11:     end if
12: end for
13: for all $v$ in $V_e$ do                                  ▷ construct hyperedge
14:     if $\mathcal{E}_c$.containHyperEdge$(e.CLASS)$ == $FALSE$ then
15:         $\mathcal{E}_c$.addHyperEdge$((e.CLASS, \emptyset))$
16:     end if
17:     $\mathcal{E}_c$.getHyperEdge$(e.CLASS)$.append$(v)$
18: end for
19: return $HG_c = \{V_c, E_c, \mathcal{E}_c, \mathbf{1.0}\}$ ▷ Each edge of $\mathcal{W}_c$ is defaulted as 1.0
```

3.2 Entity Attribute Generation

For each code entity, we generate attribute from its code snippets with a pre-trained code model. A pre-trained code model, trained on extensive corpora, learns a general-purpose code representation. Lightweight pre-trained code models extract entity attributes in the hypergraph, discovering some of the paths among AST nodes as key attributes [31] and thus capturing code semantics. This paper focuses on 6 representative pre-trained code models, including CodeBERT [21], GraphCodeBERT [25], CodeGPT [35], CodeTrans [18], CodeT5 [52], and CodeT5+ [51], frequently investigated in previous literature [57], which are illustrated in Table 1. We use lines of code as the basic unit for generating embeddings and split the field/method as a set of lines of code: $\{c_i | i = 1, 2, \cdots, k\}$. The corresponding embedding result generated by the pre-trained code model can be: $\{ebd(c_i) | i = 1, 2, \cdots, k\}$. We use the mean-pooling to aggregate embedding results of k lines of code as entity attribute: $x_i \in \mathbb{R}^d$:

$$x_i = Mean(\{ebd(c_1), \cdots, ebd(c_k)\}) \tag{3}$$

In our paper, we follow previous work [15] and heuristically set d for 768. We obtain the entity attribute space of HG_c as: $\mathbf{X}_{\mathbf{entity}} = \{x_i | i = 1, 2, \cdots, n\}$, served as the input of hypergraph learning. Feature vectors in entity attribute space are initialized as embedding results for nodes, which will further capture and learn hypergraph topology through multiple convolution layers during training.

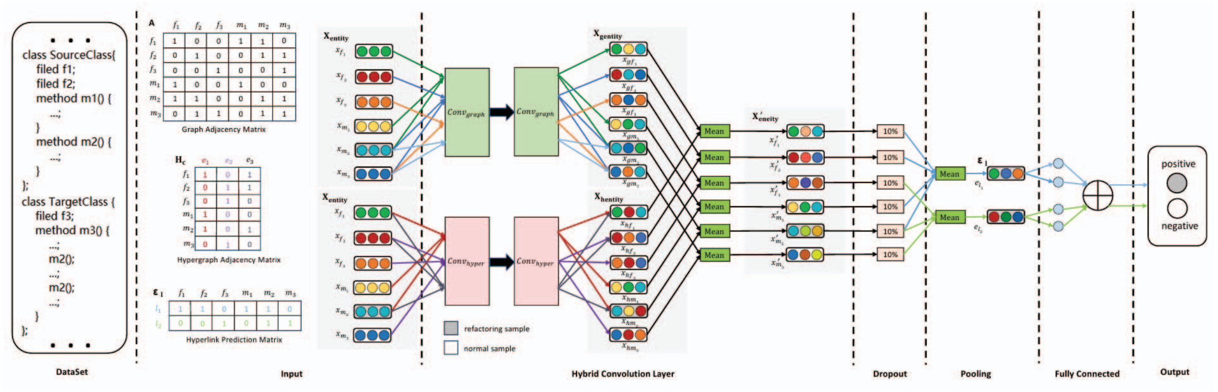
3.3 Hypergraph Learning

Hyperlink Predictor. Suggestion of Move Method refactoring opportunities can be modeled as a hyperlink prediction problem, aiming to identify a hyperlink from the hyperedge set: $\mathcal{E}_\text{l}$ that is better suited to the target method. In this paper, we set $\mathcal{E}_\text{l}$ as the original class or target class, which will be illustrated in experiment setting in Section 4. For a given potential hyperlink: l, most hyperlink prediction methods aim to learn a function Ψ such that:

$$\Psi(l) = \begin{cases} \geq \epsilon & \text{if } l \in \mathcal{E}_\text{l}, \\ < \epsilon & \text{if } l \notin \mathcal{E}_\text{l}, \end{cases} \tag{4}$$

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 hybrid method of graph neural network and hypergraph neural network for hyperlink prediction to suggest Move Method refactoring opportunities.

Graph Neural Network. Graph neural networks (GNNs) are deep learning models for handling graph data, which can learn the topology representation of nodes by propagating and aggregating neighborhood information across multiple layers of neural networks. In



```
• • •
class SourceClass{
    filed f1;
    filed f2;
    method m1() {
        ...;
    }
    method m2() {
        ...;
    }
};
class TargetClass {
    filed f3;
    method m3() {
        ...;
        m2();
        ...;
        m2();
        ...;
    }
};
• • •
```

$\mathbf{A}$	f_1	f_2	f_3	m_1	m_2	m_3
f_1	1	0	0	1	1	0
f_2	0	1	0	0	1	1
f_3	0	0	1	0	0	1
m_1	1	0	0	1	0	0
m_2	1	1	0	0	1	1
m_3	0	1	1	0	1	1

Graph Adjacency Matrix

$\mathbf{H_c}$	e_1	e_2	e_3
f_1	1	0	1
f_2	0	1	1
f_3	0	1	0
m_1	1	0	0
m_2	1	0	1
m_3	0	1	0

Hypergraph Adjacency Matrix

$\mathcal{E}_l$	f_1	f_2	f_3	m_1	m_2	m_3
l_1	1	1	0	1	1	0
l_2	0	0	1	0	1	1

Hyperlink Prediction Matrix

Figure 3. Overview of hybrid hypergraph learning framework.

this paper, we heuristically set the number of convolution layers as 2 and intensively study 3 typical graph neural networks: GCN [30], GAT [50], and GraphSAGE [27], which are frequently investigated in previous refactoring-related literature [56].

Hypergraph Neural Network. Similar to GNN, Hypergraph neural network is a deep learning model for hypergraph data, which also learns node representation through multiple convolution layers of neural networks. In our paper, we also set the number of convolution layers as 2 and investigate 3 typical hypergraph neural networks: HGNN [20], HGCN [54], and HGNN$^+$ [22] as follows:

HGNN [20] performs feature smoothing on nodes through each convolution layer as follows:

$$\mathbf{X^{l+1}} = \sigma\left(\mathbf{D_v^{-\frac{1}{2}} H W_e D_e^{-1} H^\top D_v^{-\frac{1}{2}} X^l \Theta^l}\right) \tag{5}$$

where $\mathbf{X^l}$ is the input vertex feature matrix of layer $\mathbf{l}$. $\mathbf{\Theta^l}$ is the learnable parameters of layer $\mathbf{l}$. $\mathbf{H}$ is the hypergraph incidence matrix. $\mathbf{D_e}$ is the degree matrix of hyperedges. $\mathbf{W_e}$ is the weight matrix of hyperedges. $\mathbf{D_v}$ is the degree matrix of vertex.

HGCN [54] involves aggregation information from neighboring hypernodes within hypergraphs, which can be defined as follows:

$$\mathbf{X^{l+1}} = \sigma\left(\mathbf{\tilde{D}^{-\frac{1}{2}} \tilde{H} \tilde{D}^{-\frac{1}{2}} X^l (W^l + B^l)}\right) \tag{6}$$

where $\mathbf{\tilde{H}} = \mathbf{H} + \mathbf{I}$ is the hypergraph matrix $\mathbf{H}$ with added the identity matrix $\mathbf{I}$. $\mathbf{\tilde{D}}$ is the degree matrix of $\mathbf{\tilde{H}}$.

HGNN$^+$ [22], an improved version of *HGNN*, adopts information fusion of hyperedges group to expand the hypernode information, which can be defined as follows:

$$\mathbf{X^{l+1}} = \sigma\left(\mathbf{D_v^{-1} H W_e D_e^{-1} H^\top X^l \Theta^l}\right) \tag{7}$$

The hypernode in each convolution layer involves two steps: 1) update the information of each hyperedge with hypernodes connected to itself and 2) update the information of hypernode with hyperedge connected to itself.

Hybrid HGNN-based Hyperlink Prediction. Fig. 3 presents the overview of our hypergraph learning framework including 6 layers, illustrated as follows:

Input Layer: this layer takes hyperlink prediction space: $\mathcal{E}_\mathbf{l}$, graph adjacency matrix: $\mathbf{A}$ and hypergraph adjacency matrix: $\mathbf{H_c}$, including the entity attribute space: $\mathbf{X_{entity}}$ as input.

Hybrid Convolution Layer: this layer adopts a siamese network structure, which updates $\mathbf{X_{entity}}$ through $\mathbf{Conv_{graph}}$ and $\mathbf{Conv_{hyper}}$ respectively and further fuses them. $\mathbf{Conv_{graph}}$ generates feature matrices: $\mathbf{X_{gentity}}$ from graph adjacency matrix: $\mathbf{A}$ with GCN [30], GAT [50], or GraphSAGE [27]. $\mathbf{Conv_{hyper}}$ generates feature matrices: $\mathbf{X_{hentity}}$ from hypergraph adjacency matrix: $\mathbf{H_c}$ with HGNN [20], HGCN [54], or HGNN$^+$ [22]. Then fuses $\mathbf{X_{gentity}}$ and $\mathbf{X_{hentity}}$ through mean-pooling function, which can be defined as follows:

$$\mathbf{X'_{entity}} = \mathrm{Mean}(\mathbf{X_{gentity}}, \mathbf{X_{hentity}}) \tag{8}$$

where $\mathbf{X'_{entity}}$ represents the fused result of $\mathbf{X_{gentity}}$ and $\mathbf{X_{hentity}}$.

Dropout Layer: this layer discards 10% of hidden neurons to prevent over-fitting and thus improves model's generalization ability.

Pooling Layer: this layer uses mean-pooling operation to aggregate entity attribute space: $\mathbf{X'_{entity}}$ to calculate the hyperlink prediction space: $\mathcal{E}_\mathbf{l}$. For each hyperedge $\mathbf{e} \in \mathcal{E}_\mathbf{l}$, the pooling representation: $\mathbf{e_l}$ can be calculated as follows:

$$\mathbf{e_l} = \mathrm{Mean}\left(\{\mathbf{x'_{vi}} \mid \mathbf{x_i} \in \mathrm{e} \wedge \mathbf{x'_{vi}} \in \mathbf{x'_{entity}}\}\right) \tag{9}$$

Fully Connected Layer and Output Layer: This layer converts the output of the pooling layer into a one-dimensional vector for hyperlink prediction with an activation function *sigmoid* as follows:

$$\mathbf{p_l} = sigmoid(\mathbf{W^c} \cdot \mathbf{e_l} + \mathbf{b}) \tag{10}$$

Where $\mathbf{W^c}$ is weight parameter and b is bias term. $\mathbf{p_l}$ represents the positive probability of hyperedge: $\mathbf{l}$. The cross-entropy loss function is used to train the model, which is defined as follows:

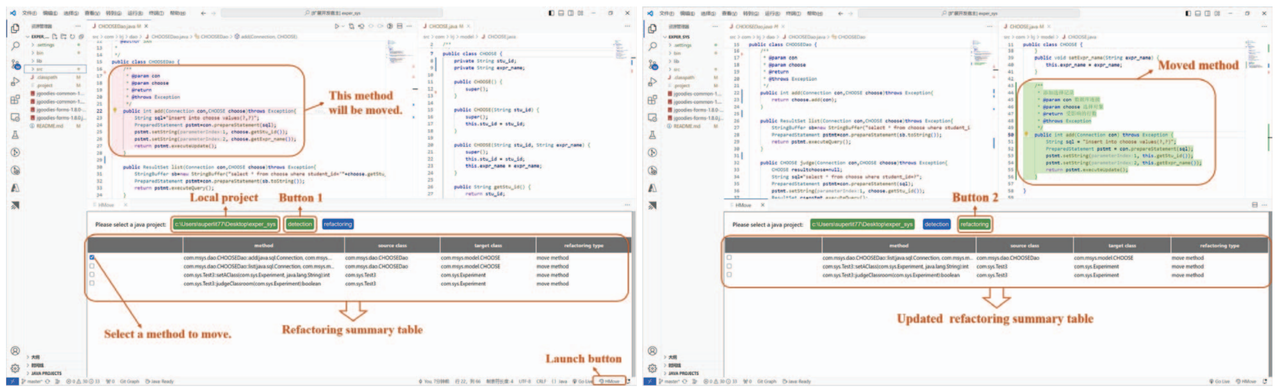
$$\mathcal{L}_{\mathbf{hyperedge}} = \frac{1}{\mathrm{N}} \sum_{\mathrm{l}} -[\mathbf{y_l} \cdot log(\mathbf{p_l}) + (1 - \mathbf{y_l}) \cdot log(1 - \mathbf{p_l})] \tag{11}$$

where $\mathbf{y_l}$ is the real label of hyperedge: $\mathbf{e}$ (1 for positive and 0 for negative). The overall training objective for hyperlink prediction includes minimizing this loss function.

To address the issue of data imbalance, we employed HyperSMOTE [6] to balance the graph data during the hypergraph construction process and further utilized the RMove's method [16] to balance the refactoring data during representation computation.

3.4 Refactoring Opportunity Suggestion

Given a subject as input, we first extract its attributed inter-class code entity dependency hypergraph and feed it into a well-trained model to generate a group of candidates. Despite large-scale training, it is still uncertain that each candidate predicted by model is valid. Thus, for each candidate, we use an LLM to verify its preconditions as a post-processing stage and return the final results.



```java
public int add(Connection con,CHOOSE choose)throws Exception{
    String sql="insert into choose values(?,?)";
    PreparedStatement pstmt=con.prepareStatement(sql);
    pstmt.setString(parameterIndex:1, choose.getStu_id());
    pstmt.setString(parameterIndex:2, choose.getExpr_name());
    return pstmt.executeUpdate();
}
```

Please select a java project: c:\Users\superlit77\Desktop\exper_sys

	method	source class	target class	refactoring type
☑	com.msys.dao.CHOOSEDao::add(java.sql.Connection, com.msys....	com.msys.dao.CHOOSEDao	com.msys.model.CHOOSE	move method
☐	com.msys.dao.CHOOSEDao::list(java.sql.Connection, com.msys.m...	com.msys.dao.CHOOSEDao	com.msys.model.CHOOSE	move method
☐	com.sys.Test3::setAClass(com.sys.Experiment, java.lang.String):int	com.sys.Test3	com.sys.Experiment	move method
☐	com.sys.Test3::judgeClassroom(com.sys.Experiment):boolean	com.sys.Test3	com.sys.Experiment	move method

```java
public int add(Connection con) throws Exception {
    String sql = "insert into choose values(?,?)";
    PreparedStatement pstmt = con.prepareStatement(sql);
    pstmt.setString(parameterIndex:1, this.getStu_id());
    pstmt.setString(parameterIndex:2, this.getExpr_name());
    return pstmt.executeUpdate();
}
```

Please select a java project: c:\Users\superlit77\Desktop\exper_sys

	method	source class	target class	refactoring type
☐	com.msys.dao.CHOOSEDao::list(java.sql.Connection, com.msys.m...	com.msys.dao.CHOOSEDao	com.msys.model.CHOOSE	move method
☐	com.sys.Test3::setAClass(com.sys.Experiment, java.lang.String):int	com.sys.Test3	com.sys.Experiment	move method
☐	com.sys.Test3::judgeClassroom(com.sys.Experiment):boolean	com.sys.Test3	com.sys.Experiment	move method

(a) A snapshot of detection view. **(b) A snapshot of refactoring view.**

Figure 4. The graphical user interface of our prototype implementation as a VSCode extension.

Trained Model Invocation: we devise a heuristic algorithm invoking the well-trained model to generate suggested results presented in Alg. 2. This algorithm takes the inter-class code entity dependency hypergraph: HG_c as input. Line 1 first initializes all the possible pairs of involved classes in V_c. Line 2 iteratively examines all the pairs and Line 3 invokes the trained model for each pair to generate the set of suggested results as $ItemList$. Each item in $ItemList$ consists of 4 elements: m_i, sc_i, tc_i, and $score_i$, representing moved method, source class, target class, and movable probability score. Line 5 examines that only a probability score greater than 0.5 will be considered. Specifically, we heuristically set the threshold as 0.5 according to previous hyperlink prediction tasks. Line 7 examines the case that $MoveMethodSet$ contains the same method as m_i having a less probability score. Line 8 will remove this item, which can further be updated. Line 12 to 13 append the result after examination. Line 18 finally returns $MoveMethodSet$ as Move Method refactoring candidates.

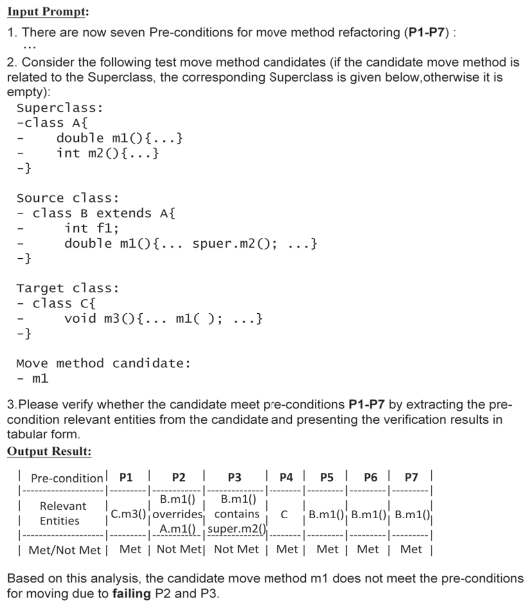
Algorithm 2 Refactoring Opportunity Suggestion Algorithm

```
Input: $HG_c = (V_c, E_c, \mathcal{E}_c, \mathcal{W}_c)$ - Inter-class Code Entity Dependency
    Hypergraph of subject.
Output: $MoveMethodSet$ - A set of refactoring suggested results.
 1: $ClassPairs \leftarrow \{(c_a, c_b) | c_a, c_b \in V_c \wedge c_a \neq c_b\}$
 2: for all $pair$ in $ClassPairs$ do
 3:     $ItemList \leftarrow$ Model($HG_c, pair$)        ▷ Invoke trained model
 4:     for all ($m_i, sc_i, tc_i, score_i$) in $ItemList$ do
 5:         if $score_i > 0.5$ then
 6:             for all ($m_t, sc_t, tc_t, score_t$) in $MoveMethodSet$ do
 7:                 if $m_i == m_t$ and $score_i > score_t$ then
 8:                     $MoveMethodSet$.remove(($m_t, sc_t, tc_t, score_t$))
 9:                     break
10:                 end if
11:             end for
12:             if $MoveMethodSet$.contain($m_i$)==FALSE then
13:                 $MoveMethodSet$.append(($m_i, sc_i, tc_i, score_i$))
14:             end if
15:         end if
16:     end for
17: end for
18: return $MoveMethodSet$
```

LLM-based Pre-Condition Verification. Refactoring operations may frequently introduce bugs and violate the consistency of program semantic [9, 17]. Following the work of Tsantalis et al. [45, 48], given a Move Method refactoring candidate, these pre-conditions before implementation should be formally verified to guarantee the behaviour preservation and functional usefulness of refactored program, which are listed as follows:

P1: The target class should not contain or inherit a method having the same signature with the moved method.
P2: The moved method should not override an abstract method or inherited method.
P3: The moved method should not contain any super method invocations.
P4: The target class should not be an interface.
P5: The moved method should not be a constructor, destructor, or delegated method.
P6: The moved method should not be synchronized.
P7: The moved method should not contain field assignments of original class.

Previous works [4, 9, 17, 39, 48] use heuristic rules/algorithms for static code analysis to verify the above pre-conditions. In most cases, these approaches rely on expert knowledge, which can be challenging to match all the extreme cases. In our paper, we use a large language model-based approach for its generalization and proficiency in code-related tasks [19]. Regarding the design of the prompt, we improve clarity, brevity, guidance, and feedback as follows: 1) providing context with sufficient information; 2) keeping prompts concise with multiple bullet points; 3) including explicit requirements and output formats in prompt; and 4) returning feedback in time to refine the thought process of LLM, to ensure its quality. Following previous work [19, 23, 46], in our paper, LLM initially learns specific terms used in pre-conditions, such as "abstract method," through examples randomly sampled from an example library [53]. Next, LLM learns from 3 high-quality and representative examples for each pre-condition along with formatted outputs. During the detection phase, LLM reviews all the pre-conditions, assesses whether each candidate meets these pre-conditions, and finally provides formatted outputs. Fig. 5 presents a running example to illustrate the used prompt. In our paper, we employ GPT-3.5 [41] and heuristically use 3 instances for each pre-condition in few-shot learning. All the used prompts and instances are available [2]. For each refactoring candidate, we filter out invalid candidates with LLM to obtain the final suggestion results.


Input Prompt:

1. There are now seven Pre-conditions for move method refactoring (**P1-P7**) :
   ...

2. Consider the following test move method candidates (if the candidate move method is related to the Superclass, the corresponding Superclass is given below,otherwise it is empty):

```
Superclass:
-class A{
-     double m1(){...}
-     int m2(){...}
-}

Source class:
- class B extends A{
-      int f1;
-      double m1(){... spuer.m2(); ...}
-}

Target class:
- class C{
-      void m3(){... m1( ); ...}
-}

Move method candidate:
- m1
```

3.Please verify whether the candidate meet pre-conditions **P1-P7** by extracting the pre-condition relevant entities from the candidate and presenting the verification results in tabular form.

Output Result:

Pre-condition	P1	P2	P3	P4	P5	P6	P7
Relevant Entities	C.m3()	B.m1() overrides A.m1()	B.m1() contains super.m2()	C	B.m1()	B.m1()	B.m1()
Met/Not Met	Met	Not Met	Not Met	Met	Met	Met	Met

Based on this analysis, the candidate move method m1 does not meet the pre-conditions for moving due to **failing** P2 and P3.

Figure 5. An illustrated example of used verification prompt.

3.5 Prototype Implementation

As illustrated in Fig. 4.(a), within the detection view, users begin by selecting the software project and clicking the "detect button" (Button 1) to generate a summary table of recommended refactoring suggestions. Each row in the table includes a checkbox, identifier of moved method, identifier of original class, and identifier of suggested target class. When an instance of refactoring suggestion is selected, the corresponding code snippet of moved method in the original class is highlighted with red color within editor. Upon selecting a refactoring suggestion and clicking the "refactoring" button (Button 2), the refactoring view in Fig. 4 will be demonstrated. The relocated method will be highlighted with green color. With the assistance of our prototype implementation, users can iteratively review all the suggestion results and apply these Move Method refactoring opportunities.

4 EVALUATION

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

RQ1: Which model combinations are the most effective in suggesting Move Method refactoring opportunities? The answer to this question would help us better understand the impact of various hypergraph neural networks on the performance of HMove.

RQ2: How accurately does HMove suggest Move Method refactoring opportunities compared to state-of-the-art refactoring tools? The answer to this question would demonstrate how well HMove performs in terms of accuracy compared to state-of-the-art refactoring tools.

RQ3: How useful does HMove suggest Move Method refactoring opportunities for developers? The answer to this question would shed insight into the usefulness of HMove in real-world applications.

Table 2: Statistics of used datasets.

Dataset	#Project	#Instance	#LOC of Moved Method
MoveTrainSet [8]	11149	163078	52
MoveAccEval [47]	11	198	28
MoveCaseEval [10]	9	10	30

4.1 Experiment Setup

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

Datasets. We train HMove on the most large-scale refactoring dataset: *MoveTrainSet* (RQ1) based on the best of our knowledge, and evaluate the accuracy of HMove with a quite high-quality dataset annotated by experts: *MoveAccEval* (RQ2). We selected several typical cases often studied in previous user studies [10] and named this dataset *MoveCaseEval* to evaluate the usefulness of HMove. Statistics of used datasets above are presented in Table 2, including involved projects (#Project), the number of Move Method refactoring instances (#Instances), and the average number of lines of code of moved methods (#LOC of Moved Method).

MoveTrainSet [8] is crawled from 11,149 projects from 3 communities: GitHub, F-Droid, and Apache, containing 163078 Move Method refactoring operations. We use *MoveTrainSet* for model training and believe it is comprehensive and diverse. *MoveAccEval* [47] contains 198 real-world Move Method refactoring oracles from 11 projects and each instance is rigorously verified by more than two experts, which is used for accuracy evaluation. *MoveCaseEval* [10]comprises 9 open-source projects and encompasses 10 typical Move Method refactoring instances, which have been continuously investigated by researchers for usefulness evaluation since 2018 [16, 32].

Evaluation Metrics. Following previous refactoring-related work [16, 31, 32], we employ three frequently used evaluation metrics: Precision, Recall, and F1-Measure to evaluate HMove, which are defined as follows:

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

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

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

Experiment Settings. We run 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 follow most of the default hyper-parameters of pre-trained models in previous work [16] and implement all the neural networks based on the Python library: PyTorch [42]. To address RQ1 (combination analysis), we implement HMove with 3 hypergraph neural networks, 3 graph neural networks and 6 pre-trained code models on *MoveTrainSet* [8], and exhaustively compare their performance of each model combination. To reduce the impact of experimental randomness, we repeat the 10-fold cross-validation process 10 times (10×10) and compute the average value of evaluation metrics as the final results. Following the previous work [16, 31], we automatically label <moved

Table 3: Comparison of various Move Method refactoring tools. ML: Machine Learning, DL: Deep Learning.

	Feature Extraction			Opportunity Detection		
Tool	Metric	Code2Vec [7]	Pre-trained Model	Heuristic	ML	DL
JDeodorant [48]	✓	✗	✗	✓	✗	✗
JMove [47]	✓	✗	✗	✓	✗	✗
PathMove [31]	✗	✓	✗	✗	✓	✗
FeDeep [33]	✓	✓	✗	✗	✗	✓
FeTruth [32]	✓	✓	✗	✓	✗	✓
LLMRefactor [46]	✗	✗	✗	✗	✗	✓
RMove [16]	✗	✓	✗	✗	✓	✓
FeGNN [56]	✓	✗	✗	✗	✗	✓
FePM [36]	✗	✗	✓	✗	✗	✓

method, source class> and <moved method, target class> as positive and negative samples respectively to address the imbalance issue.

To address RQ2 (accuracy evaluation), we select the best model combinations according to the results of RQ1 and evaluate their performance on *MoveAccEval* [47] and compare these results to 9 state-of-the-art Move Method refactoring tools including LLM-Refactor [46], FeTruth [32], FeDeep [33], PathMove [31], JDeodorant [48], JMove [47], RMove [16], FeGNN [56], and FePM [36]. Table 3 presents the mechanic comparison of these state-of-the-art refactoring tools, which vary in feature extraction and opportunity detection. We configure these tools with default parameters and prompts. We also 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 [2].

To address RQ3 (usefulness evaluation), we conduct a user study following the methods outlined in [12, 16]. We employed a random sampling approach to recruit 50 voluntary participants from our collaborative companies who are interested in code refactoring detection tools for our user study. All the participants are not the authors of this paper. They are invited to analyze the suggested results of 3 selective tools, including FeTruth [32], RMove [16], and the best combination of HMove in RQ1 and RQ2 on *MoveCaseEval* [10]. FeTruth and RMove are investigated for they receive relatively high ratings from users according to recent work [16, 32]. Following previous work [16], this user study did not involve any specific tasks. Participants reviewed the detection results of various tools and further completed a feedback questionnaire regarding their personal refactoring experiences with each refactoring tool. Finally, we collect and analyze the questionnaire results which are included in [2].

4.2 Experiment Result

Combination Analysis (RQ1). We conduct a systematic comparison of HMove on 54 combinations of 3 hypergraph neural networks, 3 graph neural networks and 6 pre-trained code models. Table 4 presents the performance of these model combinations on *MoveTrainSet* [8], where rows and columns are labeled by combination of hypergraph neural network and graph neural network, and pre-trained code models respectively. For each column, we highlight the greatest precision, recall, and f1-measure scores with a gray background. Furthermore, we highlight the greatest scores of all possible model combinations with a "*" mark. As presented in Table 4, we observed that GCN+HGNN, GraphSAGE+HGNN, and GraphSAGE+HGNN$^+$ outperform other model combinations on precision, recall, and f1-measure respectively. Specifically, GCN+HGNN+CodeT5 achieves the highest precision score. GraphSAGE+HGNN+CodeT5 achieves the highest recall score and GraphSAGE+HGNN$^+$+CodeT5 achieves the highest f1-measure score.

To verify the significance of the above model combinations on experimental results, we further perform effect size: Cliff's δ [14], a non-parametric effect size, to check whether a statistically significant difference exists between these combinations and other combinations. The values range from negligible ($|\delta| < 0.147$), small ($0.147 \le |\delta| < 0.33$), medium ($0.33 \le |\delta| < 0.474$), and large ($|\delta| \ge 0.474$). We present Cliff's δ of combinations: GCN+HGNN, GraphSAGE+HGNN, and GraphSAGE+HGNN$^+$ with other combinations on precision, recall, and f1-measure in Table 5. As presented, we observed a significant difference in precision, recall, and f1-measure score of these combinations compared to other combinations. A possible explanation is that these model combinations make it easier to learn the characteristics of hypergraphs and features generated from pre-trained code models, and thus perform better. We additionally performed the Mann-Whitney U test to investigate the statistical differences between the various combinations. As shown in Table 5, all combinations marked with ** demonstrate statistically significant differences, with p-values below 0.05. These experimental results also present a significant difference.

Answer to RQ1: The most effective combinations include GCN+HGNN+CodeT5, GraphSAGE+HGNN+CodeT5, and GraphSAGE+HGNN$^+$+CodeT5.

Accuracy Evaluation (RQ2). To evaluate the accuracy of HMove, we selected the three most effective model combinations from RQ1–GCN+HGNN+CodeT5, GraphSAGE+HGNN+CodeT5, and GraphSAGE+HGNN$^+$+CodeT5–labeling them as HMove-α, HMove-β, and HMove-γ, respectively. Furthermore, we apply trained neural networks on these combinations and evaluate their performance on *MoveAccEval* [47]. We compare HMove with 9 state-of-the-art Move Method refactoring tools including LLMRefactor [46], FeTruth [32], FeDeep [33], PathMove [31], JDeodorant [48], JMove [47], RMove [16], FeGNN [56], and FePM [36].

Table 6 presents the accuracy of each refactoring tool on each subject of *MoveAccEval* [47]. For each subject, we highlight the greatest precision, recall, and f1-measure score with a gray background. Specially, for the row: Avg, we highlight the greatest precision, recall, and f1-measure score of 9 state-of-the-art Move Method refactoring tools with red color and gray background. We also highlight the greatest score of HMove-α, HMove-β and HMove-γ with blue color and gray background. As presented in Table 6, we observed that 1) RMove [16] demonstrates a higher recall and f1-measure score, while FePM [36] achieves a greater precision score. We use the precision of FePM [36], and the recall and f1-measure score of RMove [16] as baseline. 2) Compared to the best results of state-of-the-art refactoring tools, HMove-α achieves the greatest precision score and presents an increase of 27.8%. HMove-β achieves the greatest recall score and presents an increase of 2.5%. HMove-γ achieves the greatest f1-measure score and presents an increase of 18.5%. 3) HMove outperforms state-of-the-art refactoring tools on most subjects. In a few cases, state-of-the-art refactoring tools present a slightly higher recall score than HMove. The accuracy improvement of HMove can be attributed to effective capture of

Table 4: Performance of various model combinations on *MoveTrainSet* [8]. P:Precision, R:Recall, F1:F1-measure.

Model Combination	CodeBERT [21]			CodeGPT [35]			CodeT5 [52]			CodeT5+ [51]			CodeTrans [18]			GraphCodeBERT [25]		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
GAT+HGNN	0.706	0.364	0.480	0.785	0.460	0.580	0.810	0.667	0.731	0.771	0.409	0.535	0.881	0.596	0.711	0.886	0.551	0.679
GAT+HGCN	0.692	0.556	0.616	0.864	0.621	0.722	0.888	0.687	0.775	0.895	0.610	0.726	0.887	0.631	0.738	0.852	0.611	0.712
GAT+HGNN$^+$	0.752	0.139	0.235	0.788	0.525	0.630	0.864	0.576	0.691	0.733	0.595	0.657	0.651	0.273	0.384	0.869	0.544	0.669
GCN+HGNN	**0.906**	0.318	0.471	**0.918**	0.462	0.614	**0.947***	0.631	0.758	**0.908**	0.596	0.720	**0.936**	0.662	0.775	**0.894**	0.605	0.722
GCN+HGCN	0.782	0.434	0.558	0.857	0.615	0.716	0.753	0.674	0.711	0.800	0.621	0.699	0.892	0.500	0.641	0.866	0.586	0.699
GCN+HGNN$^+$	0.834	0.623	0.713	0.876	0.580	0.698	0.831	0.571	0.677	0.876	0.613	0.721	0.867	0.626	0.727	0.811	0.586	0.680
GraphSAGE+HGNN	0.765	**0.688**	0.724	0.828	**0.707**	0.763	0.732	**0.854***	0.788	0.821	**0.708**	0.760	0.897	**0.718**	0.798	0.780	**0.662**	0.716
GraphSAGE+HGCN	0.782	0.313	0.447	0.826	0.601	0.696	0.898	0.621	0.734	0.887	0.475	0.618	0.866	0.644	0.738	0.877	0.576	0.695
GraphSAGE+HGNN$^+$	0.894	0.682	**0.774**	0.899	0.687	**0.779**	0.897	0.748	**0.815***	0.899	0.672	**0.769**	0.922	0.717	**0.807**	0.882	0.615	**0.725**

Table 5: Cliff's δ and Mann-Whitney U test between selective combinations and other combinations on precision, recall, and f1-measure score.

Model Combination	GCN+HGNN Precision	GraphSAGE+HGNN Recall	GraphSAGE+HGNN$^+$ F1-Measure
GAT+HGNN	1** (large)	0.94** (large)	0.94** (large)
GAT+HGCN	0.94** (large)	0.94** (large)	0.72** (large)
GAT+HGNN$^+$	1** (large)	1** (large)	1** (large)
GCN+HGNN	—	0.97** (large)	0.78** (large)
GCN+HGCN	1** (large)	0.94** (large)	1** (large)
GCN+HGNN$^+$	1** (large)	1** (large)	0.94** (large)
GraphSAGE+HGNN	0.94** (large)	—	0.44 (medium)
GraphSAGE+HGCN	0.94** (large)	1** (large)	0.89** (large)
GraphSAGE+HGNN$^+$	0.58** (large)	0.33 (medium)	—

[1] ** indicates significant results from the Mann-Whitney U test, with a p-value less than 0.05.

Move Method refactoring characteristics with hypergraph neural network. Software practitioners can consider HMove-γ as a reliable tool to suggest Move Method refactoring opportunities. We will select HMove-γ for usefulness evaluation in RQ3.

Answer to RQ2: HMove demonstrates an increase of 27.8% in precision, 2.5% in recall and 18.5% in f1-measure compared to state-of-the-art refactoring tools.

Usefulness Evaluation (RQ3). To evaluate the usefulness of HMove, we conduct a user study of 50 industrial engineers to review the suggestion results of 3 representative refactoring tools including RMove[16], FeTruth[32] and HMove-γ for each instance in *MoveCaseEval* [11]. To guarantee that participants are unaware of which tool we developed, for each instance, we blindly display suggested results of these refactoring tools in a shuffled order to each participant. Participants will evaluate whether the method should be moved and whether the target class to receive moved method is appropriate. After reviewing, participants are encouraged to answer Q1-Q4 in Table 8. Details of all the questionnaire results are available [2]. As illustrated in Figure 6, our questionnaire mixed of multiple types of questions, including 4 multiple-choice questions and 1 open-ended question. The design of the questionnaire is based on insights from previous studies [12, 16], and it primarily examines developers' refactoring practices and their experiences with three distinct refactoring tools.

Fig. 6 presents the results of our survey. 40% of participants reported having 4 to 5 years of experience in software development, while 44% frequently considered Move Method refactoring during development. Additionally, 24% indicated that Move Method refactoring can be performed without the aid of tools. Upon reviewing the results of various refactoring tools, 68% of participants recommended HMove compared to other refactoring tools.

We further analyze each instance of *MoveCaseEval* [11] in depth to demonstrate the difference between HMove and other refactoring tools in practice. These instances, chosen for their comprehensiveness and diversity, have been frequently studied in previous literature [16, 47], and were selected to systematically compare the detection effectiveness of various refactoring tools. Table 9 presents the suggested results of RMove [16], FeTruth [32], and HMove-γ compared to results labeled by experts. For refactoring tools, we use "Suggested" and "MoveCorrect" to evaluate whether this method is detected to be moved and further correctly moved to the target class in each instance. We use ✓ and × to highlight the results. As presented in Table 9, we observed that RMove [16] suggests 7 methods accurately and 5 of them are correctly moved. FeTruth [32] suggests 6 methods accurately and 3 of them are correctly moved. In comparison, HMove-γ suggests 9 methods accurately and 7 of them are correctly moved. HMove can provide more comprehensive and precise results than FeTruth [32] and RMove [16].

As presented in Table 9, in samples 1, 3, 6, 7, 9, and 10, HMove and at least one of RMove and FeTruth successfully identified moved methods and locations. In sample 2, HMove successfully identified moved methods while both of RMove and FeTruth failed. A possible reason is that limited dependencies between JFreeChart and Title cause challenges for RMove and FeTruth to identify while HMove can effectively learn features from code dependencies and semantics together and thus perform better. In sample 4, in addition to moved methods, HMove successfully identified moved locations while RMove and FeTruth failed. The possible reason can be explained as sample 4. In sample 5, the reverse applies. HMove and FeTruth failed to identify moved locations while RMove succeeded. A possible explaination is that RMove captures AST-related features, aiding in discovering the correct moved location. In sample 8, HMove, RMove, and FeTruth all failed. A possible reason is that the target class has a few dozen lines of code and limited dependency with other classes, causing challenges for feature envy identification with tools. Overall, these success and failure cases will shed lights on how to improve our tool in future work.

Answer to RQ3: HMove is more practical compared to state-of-the-art refactoring tools for 68% of participants.

5 DISCUSSION

In this section, we discuss the threats, limitations to validity, 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

Table 6: Accuracy of various refactoring tools on each subject of *MoveAccEval* [47]. P: Precision, R: Recall, F1: F1-Measure.

	LLMRefactor [46]			FeTruth [32]			FeDeep [33]			PathMove [31]			JDeodorant [48]			JMove [47]		
Subject	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Ant	0.214	0.200	0.207	0.446	0.691	0.542	0.143	0.133	0.138	0.197	0.560	0.292	0.171	0.480	0.252	0.173	0.840	0.287
Derby	0.471	0.444	0.457	0.381	0.742	0.503	0.118	0.111	0.114	0.134	0.364	0.196	0.163	0.460	0.241	0.104	0.768	0.183
DrJava	0.250	0.263	0.256	0.303	0.833	0.444	0.118	0.105	0.111	0.428	0.500	0.461	0.128	0.555	0.208	0.128	0.777	0.220
JFreeChart	0.267	0.250	0.258	0.382	0.712	0.497	0.222	0.250	0.235	0.384	0.541	0.449	0.121	0.563	0.199	0.276	0.544	0.366
JGroups	0.273	0.250	0.261	0.246	0.837	0.380	0.091	0.283	0.138	0.243	0.410	0.305	0.244	0.417	0.308	0.106	0.772	0.186
JHotDraw	0.270	0.263	0.267	0.378	**0.833**	0.520	0.206	0.184	0.194	0.532	0.422	0.471	0.139	0.560	0.223	0.147	0.646	0.240
JTopen	0.368	0.350	0.359	0.357	0.774	0.489	0.158	0.150	0.154	0.416	0.512	0.459	0.207	0.447	0.283	0.208	**0.894**	0.337
JUnit	0.563	0.500	0.529	0.461	0.814	0.589	0.211	0.222	0.216	0.348	0.444	0.390	0.102	0.611	0.175	0.074	0.820	0.136
Lucene	0.333	0.300	0.316	0.471	0.791	0.590	0.125	0.100	0.111	0.346	0.500	0.409	0.236	0.250	0.243	0.202	0.742	0.318
MvnForum	0.357	0.313	0.333	0.354	0.786	0.488	0.267	0.250	0.258	0.264	0.389	0.315	0.142	0.542	0.225	0.217	0.720	0.333
Tapestry	0.286	0.267	0.276	0.323	**0.813**	0.462	0.286	0.267	0.276	0.304	0.517	0.383	0.148	0.330	0.204	0.112	0.614	0.189
Avg	0.332	0.309	0.320	0.373	0.784	0.501	0.177	0.187	0.177	0.327	0.469	0.375	0.164	0.474	0.233	0.159	0.740	0.254

	RMove [16]			FeGNN [56]			FePM [36]			HMove-α			HMove-β			HMove-γ		
Subject	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Ant	0.491	**0.905**	0.637	0.462	0.614	0.527	0.721	0.482	0.578	**0.975**	0.666	**0.791**	0.647	0.733	0.688	0.941	0.600	0.733
Derby	0.484	0.825	0.610	0.500	0.689	0.579	0.576	0.369	0.450	**0.983**	0.722	0.833	0.680	**0.944**	0.791	0.944	**0.944**	**0.944**
DrJava	0.526	**0.960**	0.680	0.376	0.762	0.504	0.686	0.442	0.538	**0.964**	0.667	0.788	0.630	0.895	0.739	0.938	0.790	**0.857**
JFreeChart	0.472	0.857	0.609	0.341	0.614	0.438	0.633	0.414	0.501	**0.909**	0.684	0.781	0.667	**0.875**	0.757	0.867	0.813	**0.839**
JGroups	0.521	0.760	0.618	0.276	0.801	0.411	0.617	0.321	0.422	**0.889**	0.625	0.734	0.667	**0.952**	0.784	0.786	0.917	**0.846**
JHotDraw	0.517	0.694	0.593	0.476	0.671	0.557	0.650	0.467	0.544	**0.974**	0.368	0.535	0.763	0.763	0.763	0.875	0.737	**0.800**
JTopen	0.484	0.872	0.622	0.521	0.693	0.595	0.701	0.442	0.542	**0.963**	0.550	0.700	0.696	0.800	**0.744**	0.929	0.450	0.606
JUnit	0.524	0.853	0.649	0.480	0.643	0.550	0.646	0.387	0.484	0.895	**0.944**	0.919	**0.944**	0.917	**0.930**	0.857	0.667	0.750
Lucene	0.496	0.812	0.616	0.453	0.648	0.533	0.616	0.394	0.481	0.900	0.727	0.804	**0.909**	**0.909**	**0.909**	0.769	**0.909**	0.833
MvnForum	0.505	0.772	0.611	0.262	0.758	0.389	0.651	0.350	0.455	0.917	0.688	0.786	0.762	0.944	0.843	**0.941**	**1.000**	**0.970**
Tapestry	0.469	0.810	0.594	0.402	0.674	0.504	0.704	0.352	0.469	0.900	0.600	0.720	0.833	0.667	**0.741**	**1.000**	0.533	0.696
Avg	0.499	**0.829**	**0.622**	0.414	0.688	0.508	**0.655**	0.631	0.497	**0.933**	0.658	0.763	0.745	**0.854**	0.790	0.895	0.760	**0.807**

Table 7: Comparison of various large language models.

Models	% of filtered candidates	ΔP	ΔR	ΔF1
CodeLlama (7B) [44]	18.5%	0.178 ↑	0.213 ↑	0.198 ↑
DeepSeek-Coder (7B) [26]	18.1%	0.173 ↑	0.242 ↑	0.213 ↑
GPT-3.5 [5]	**19.8%**	**0.196 ↑**	**0.291 ↑**	**0.249 ↑**
GPT-4 [41]	19.4%	0.187 ↑	0.284 ↑	0.242 ↑

Table 8: Questionnaire of Move Method refactoring tool.

Participants Information

Q1. Years of Experience in Software Development
☐ Less than 1 year ☐ 1 – 3 years ☐ 4 – 5 years ☐ 6 – 10 years ☐ More than 10 years

Tools Evaluation

Q2. How often do you consider using Move Method Refactoring in your projects?
☐ Never ☐ Rarely ☐ Sometimes ☐ Often ☐ Always

Q3. Would you prefer to use tools for detecting Move Method Refactoring opportunities?
☐ Yes ☐ Maybe ☐ No

Q4. Which of these tools do you recommend? (Please select only one)
☐ Tool 1 ☐ Tool 2 ☐ Tool 3

further keep reviewing the training data and filtering the noise. The second threat exists in the pre-condition verification. 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 HMove. Fig. 7 presents the average run-time overhead of various refactoring tools for each subject on *MoveAccEval* [47]. We observed that HMove responds within 2.27s on average. In comparison, HMove merely outperforms 3 state-of-the-art refactoring tools. We will further optimize and evaluate the run-time of HMove on more datasets. The fourth threat is the

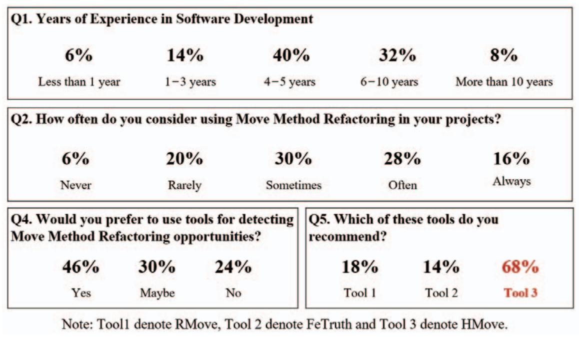


Figure 6. Questionnaire results achieved in our user study.

hybrid design of our neural network. We present our ablation experiment on [2] due to limited space and the results suggest that each has a significant impact on performance.

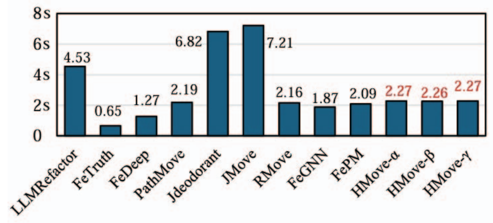


Figure 7. Average run-time overhead of various refactoring tools for each subject on *MoveAccEval* [47].

Limitations. The first limitation comes from hyper-parameters. We follow the default hyper-parameters in previous literature and will further investigate the impact of hyper-parameters on performances. Second, our tool relies on existing implementations of

Table 9: Suggested result of various Move Method refactoring tools for each instance in *MoveCaseEval* [11].

					RMove		FeTruth		HMove-γ	
	Subject	Moved Method	Source Class	Target Class	Suggested	MoveCorrect	Suggested	MoveCorrect	Suggested	MoveCorrect
1	JFreeChart	createLegendItemBlock	LegendTitle	LegendItem	✓	✓	✓		✓	✓
2	JFreeChart	drawTitle	JFreeChart	Title					✓	
3	JGroups	sendNotification	NotificationBus	Promise			✓	✓	✓	✓
4	Derby	getInputStream	NetAgent	NetConnectionReply	✓		✓		✓	✓
5	DrJava	findPrevious	FindReplacePanel	FindReplaceMachine	✓	✓	✓		✓	
6	JTopen	readObject	HTMLTree	HTMLVector	✓	✓			✓	✓
7	Ant	XsetIgnore	MatchingTask	FileSet	✓	✓			✓	✓
8	Tapestry	defaultModel	Select	RenderDisabled						
9	Lucene	getDrillDownAccumulator	DrillSideways	FacetSearchParams	✓		✓	✓	✓	✓
10	MvnForum	addJiveThreadWatch	JiveThreadXML	ThreadXML	✓	✓	✓	✓	✓	✓

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's interpretability and further investigated more interpretable approaches. Fourth, our tool uses LLM as a main component while we didn't investigate the impact of prompt length and LLM hallucination, which will be fully explored in our future work. Additionally, we investigated 4 SOTA LLMs, including Code Llama [44], DeepSeek [26], GPT-4 [5], and GPT-3.5 [41], and evaluated their impact on the performance of HMove-γ on *MoveAccEval* [47]. As presented in Table 7, all the large language models filtered approximately 20% of candidates as invalid candidates. After calculating the ΔP, ΔR, and $\Delta F1$ score between filtering and not filtering invalided candidates, we observed GPT-3.5 exhibited a more comprehensive and stable performance compared to the other three LLMs. Fifth, we solely use lexical match-based metrics such as precision, recall, and F1-measure and don't test the semantic correctness of the generated solution after refactoring. We preliminarily sampled 100 candidates from *MoveAccEval* [47] and measured the compilation success rate. The results are promising with almost 99%. Detailed results are available in our public data repository [2]. We will further employ more semantic-related metrics to evaluate in our future work.

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 feature envy-related refactoring approaches like Move Field refactoring and Extract+Move Method refactoring. 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 feature envy-related refactoring tasks.

6 RELATED WORK

We introduce related techniques that motivate our work as follows:

Heuristic-based approaches. The most representative work is JDeodorant introduced by Tsantalis et al. [48]. JDeodorant detects Feature Envy by measuring the distance between moved method and target class. To ensure the refactoring correctness, JDeodorant also employs a set of pre-conditions and automatically verifies them. Terra et al. [47] proposed JMove based on the similarity of established dependency sets. The results show that JMove outperforms JDeodorant, especially for large methods. Liu et al. [34] introduced Domino based on a heuristic that similar methods should be moved together. Ujihara et al. [49] introduced C-JRefRec by computing the semantic similarity between moved method and target class with TF-IDF. Bavota et al. [10] introduced MethodBook considering both method calls and identifier information with Relational Topic Model (RTM) to detect feature envy.

Machine learning-based approaches. Liu et al. [33] are the first to utilize deep learning to detect feature envy. They generate implicit representations with word2vec [40] from historical samples and feed them into Convolutional Neural Network (CNN) model to conduct binary classification. Liu et al. [32] further used filtered real-word open-source examples as training dataset. The experimental results showed that the real-world dataset can greatly improve the effectiveness of model. Kurbatova et al. [31] employed Code2Vec [7] to extract features and trained Support Vector Machine (SVM) to detect feature envy. Cui et al. [16] employed code embedding techniques and graph embedding techniques to extract structural and semantic features of code snippets respectively, and fuse them into machine learning classifiers for feature envy detection. Yu et al. [56] collected code metrics and calling relationships to construct attributed graph and employed Graph Neural Network for training and detecting feature envy. Ma et al. [37] employed pre-trained code model to represent methods or classes as embeddings and fed them into deep learning classifiers to detect feature envy.

Compared to previous work, our approach introduces the concept of hypergraph, which enables the analysis of code dependencies binding multiple code elements. Our approach introduce LLM, which supports the flexible extensions of more verification pre-conditions.

7 CONCLUSION

In this paper, we propose HMove, a novel hypergraph learning-based method for Move Method refactoring opportunity suggestions. Our approach utilizes a hybrid hypergraph neural network alongside a large language model (LLM) to build a refactoring suggestion system. The findings demonstrate that HMove showed a 27.8% increase in precision, a 2.5% increase in recall, and an 18.5% increase in f1-measure over leading tools with 68% of participants finding it more practical for real-world applications. Overall, HMove advances Move Method refactoring by integrating hypergraph neural network with large language model, marking a significant step forward in refactoring tools.

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, 62472341, 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] 2015. *Commit of jline2.* https://github.com/jline/jline2/commit/1eb3b624b288a4b1a054420d3efb05b8f1d28517

[2] 2024. *Data Link.* https://anonymous.4open.science/r/phmove-5FBC

[3] 2024. *jline2.* https://github.com/jline/jline2

[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] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[6] 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.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[8] 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.

[9] 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.

[10] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2013. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2013), 671–694.

[11] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694.

[12] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. 2016. Using cohesion and coupling for software remodularization: Is it enough? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 1–28.

[13] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005* (2013).

[14] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis.* Psychology Press.

[15] 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.

[16] 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.

[17] 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.

[18] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).

[19] Sidong Feng and Chunyang Chen. 2024. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[20] 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.

[21] 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

[22] 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.

[23] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[24] Google BigQuery. [n.d.]. Google BigQuery. https://console.cloud.google.com/marketplace/details/github/github-repos. Accessed: 2021.

[25] 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*.

[26] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[27] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[28] 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).

[29] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. 2019. ENRE: A Tool Framework for Extensible eNtity Relation Extraction. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 67–70. https://doi.org/10.1109/ICSE-Companion.2019.00040

[30] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[31] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 315–322.

[32] Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 908–920.

[33] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* (2019).

[34] Hui Liu, Yuting Wu, Wenmei Liu, Qiurong Liu, and Chao Li. 2016. Domino effect: Move more methods once a method is moved. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 1–12.

[35] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[36] Wenhao Ma, Yaoxiang Yu, Xiaoming Ruan, and Bo Cai. 2023. Pre-trained Model Based Feature Envy Detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 430–440.

[37] Wenhao Ma, Yaoxiang Yu, Xiaoming Ruan, and Bo Cai. 2023. Pre-trained Model Based Feature Envy Detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 430–440.

[38] Vadim Markovtsev and Waren Long. 2018. Public git archive: a big code dataset for all. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 34–37.

[39] 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.

[40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[41] OpenAI. [n.d.]. Introducing ChatGPT. https://openai.com/blog/chatgpt/. Accessed: July 3, 2023.

[42] 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.

[43] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016), 731–747.

[44] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[45] 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.

[46] 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).

[47] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. 2018. JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138 (2018), 19–36.

[48] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[49] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. c-JRefRec: Change-based identification of Move Method refactoring opportunities. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 482–486.

[50] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.

[51] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[52] 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).

[53] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2021. Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Comput. Surv.* 53, 3 (2021), 63:1–63:34. https://doi.org/10.1145/3386252

[54] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. 2019. Hypergcn: A new method for training graph convolutional networks on hypergraphs. *Advances in neural information processing systems* 32 (2019).

[55] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*. 1693–1703.

[56] Dongjin Yu, Yihang Xu, Lehui Weng, Jie Chen, Xin Chen, and Quanxin Yang. 2022. Detecting and Refactoring Feature Envy Based on Graph Neural Network. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 458–469.

[57] 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.