DOI:10.1587/transinf.2023EDP7192

Publicized:2024/05/14

Paper

Large Class Detection using GNNs: A graph based deep learning approach utilizing three typical GNN model architectures

HanYu Zhang [†], *Nonmember and* **Tomoji Kishi**[††], *Nonmember*

SUMMARY Software refactoring is an important process in software development. During software refactoring, code smell is a popular research topic that refers to design or implementation flaws in the software. Large class is one of the most concerning code smells in software refactoring. Detecting and refactoring such problem has a profound impact on software quality. In past years, software metrics and clustering techniques have commonly been used for the large class detection. However, deep-learning-based approaches have also received considerable attention in recent studies. In this study, we apply graph neural networks (GNNs), an important division of deep learning, to address the problem of large class detection. First, to support the extensive data requirements of the deep learning task, we apply a semiautomatic approach to generate a substantial number of data samples. Next, we design a new type of directed heterogeneous graph (DHG) as an input graph using the methods similarity matrix and software metrics. We construct an input graph for each class sample and make the graph classification with GNNs to identify the smelly classes. In our experiments, we apply three typical GNN model architectures for large class detection and compare the results with those of previous studies. The results show that the proposed approach can achieve more accurate and stable detection performance.

key words: *Software Refactoring, Code Smell, Large Class, Deep Learning, Graph neural networks.*

1. Introduction

Software Refactoring plays an important role in software development and maintenance. It can optimize the internal code or structure of software without changing its external behavior [1][2]. Code Smell is one of the most important focuses of software refactoring, and refers to flaws in software design or implementation. The term was first mentioned by Kent Beck and further elaborated in the publication by Fowler [2] in 1992.

Large Class is one of the most investigated code smells, which refers to classes with too many fields or methods [2]. The presence of a large class significantly reduces the readability, maintainability, and reusability of the software while also violating one of the basic design principles of object-oriented development (OOD): single responsibility principle (SRP). The large class has also received extensive attention in the research field, where it is also well-known as the God Class or Blob [5]. According to a survey conducted by Sharma [3] and AbuHassan [4], the number of studies on large classes has always been greater than the number of studies on other code smells in the past years.

† The author is with Inner Mongolia University of Science & Technology, Inner Mongolia, Baotou, 014010 China.
†† The author is with Waseda University, Shinjuku, Tokyo, 169-8555 Japan.

From past studies on large class detection, approaches based on software metrics have been traditionally and commonly used. For instance, Lanza [6] introduced a formula with three software metrics and set thresholds for each to identify large classes. Such a metric-based approach could be useful for many refactoring situations. However, metric-based approaches rely heavily on metrics and thresholds defined by developers. It is also challenging to find the most appropriate metrics because different metrics can yield different detection results.

To avoid the manual design by researchers, the approach based on machine learning techniques has also received much attention. Fontana et al. [7] applied 16 machine learning algorithms to large class detection and horizontally compared the results. The approach performed a large experiment by applying machine learning algorithms to each code smell and proved the effectiveness of machine learning in large class detection.

Another machine learning-based approach uses clustering algorithms to identify extracted class opportunities from the target class. For example, Akash [12], used the hierarchical agglomerative clustering (HAC) algorithm to identify extracted class opportunities. Classes identified as having extraction opportunities were marked as large classes.

Previous studies demonstrated the feasibility of machine learning techniques for large class detection. However, existing machine-learning-based approaches still have some limitations, such as feature selection and threshold design. In recent studies, deep learning methodologies have also been applied to large class detection.

Compared with the above two types of machine learning-based approaches, the deep learning approach achieved better performance without feature engineering. However, this always requires a substantial number of training data samples, and the design of the neural network is also a critical consideration. Liu et al. [8] first introduced an automatic dataset-generation approach called smell introduction refactoring. This approach involves performing unwanted refactoring, which reduces the software quality, to generate a training dataset with a large number of data samples. Using the automatically generated dataset, they constructed a composite network by combining LSTM with dense layers. Subsequently, a classifier is trained to detect large classes using metrics and textual information as input data. Although this approach effectively harnessed deep learning techniques for large class detection, it had limitations, primarily because of the simplicity of the network design and the quality of the dataset. Further research is needed to improve detection performance.

In this study, we applied an important division of deep learning: GNNs, to large class detection. We used GNNs because of two primary factors. First, as also mentioned in the existing deep learning approach [8], compared with traditional statistical machine learning techniques, deep learning techniques could help us select the most useful features and establish complex correspondences from input to output. Second, in the existing studies on code smell detection, representing the input data as a graph has also been a commonly used technique [4]. We assume that integrating graph representation techniques with deep learning in large class detection may yield even more favorable outcomes.

To obtain sufficient high-quality data samples for the deep learning task, we applied a semiautomatic approach to generate a dataset with a large number of data samples. Next, we created a new type of DHG as the input graph, based on the method similarity matrix and 22 types of software metrics as the node features. Then, we constructed an input graph for each data sample and considered the detection process as a graph classification task. Finally, we performed evaluation experiments using a dataset that was manually reviewed and compared the results with those of existing studies. The results show that the proposed approach achieves more accurate and stable detection performance.

This paper is a further extension of our previous study [29]. Compared with the previous study, which utilized GCN for long method detection, we encountered distinct challenges in the realm of large class detection. Firstly, the dataset generation approaches used in the previous study [29] must be reconsidered and redesigned, as the basic concepts (such as: metrics, rules) are no longer suitable for large class detection. Secondly, it is necessary to design a new type of class-level input graph for GNN applications. Thus, in this study, we made the following expansions.

- First, we redesigned the semiautomatic dataset generation approach from our previous study [29] for large class detection. To achieve this, we first define two types of class merging opportunities that can be automatically detected to generate positive class samples. Next, we use three types of class-level metrics as the basis for categorizing the class samples and define the corresponding grouping rules for semiautomatic dataset generation. Furthermore, to reduce manual labeling costs, we developed a new assistive labeling tool to help developers quickly label large numbers of sample classes. This tool can be found at https://github.com/Bankzhy/lclb.
- Second, to apply GNNs to the task of large class detection, we first built a similarity matrix based on the structural and conceptual similarities of all methods. Based on the similarity matrix, a new type of DHG was designed as the input graph. Furthermore, we reselected 11 metrics as node features at each level of the method and class.
- Finally, in addition to the basic modeling architecture GCN, we further applied two other modeling architectures: GraphSage and GAT, to large class detection and made a horizontal comparison of their detection performance with existing approaches [8][12].

The remainder of this paper is organized as follows. In Section 2, several studies on large class detection are introduced. In Section 3, we elaborate on the proposed approach including: dataset generation, metric calculations, graph construction, and the GNN for large class detection. In Section 4, we evaluate the performance of the proposed approach and compare the results with those of existing large class detection approaches. Finally, the conclusions of our approach are presented in Section 5.

2. Related Work

Several large class detection approaches have been proposed over the past few decades. In terms of the techniques used by developers, metric-based and machine learning-based approaches are the two main approaches.

The metric-based approach begins by calculating a series of metrics to capture the software characteristics. Using such metrics, developers can identify a large class by using predefined calculation formulas and thresholds. In the study proposed by Lanza [6], they used three metrics: access to foreign data (ATFD), weighted method count (WMC) and tight class cohesion (TCC) to identify the large class following Eq.1, in which ATFD is a software metric that represents the number of external classes from which a given class accesses attributes directly or via accessory methods. The WMC is the sum of the statistical complexities of all the methods in a class, and TCC is the relative number of methods that are directly connected via attribute access.

$$ATFD > FEW \wedge WMC > VERY\ HIGH \wedge \\ TCC < ONE\ THIRD \tag{1}$$

DÉCOR, proposed by Moha [9] gives the following briefer formula: $NOM + NOA > VERY\ HIGH$, in which the number of methods (NOM) and number of attributes (NOA) represent the number of methods and fields in the target class, respectively.

In metric-based detection approaches, metrics and predefined thresholds are the key points in determining whether a class can be identified as a large class. However, these metrics reflect only specific software features. It is also difficult to determine optimal features or thresholds. In recent years, machine learning-based approaches have received increasing attention in code smell detection. In 2016, Fontana [7] provided a statistical machine learning-based approach to detect large classes. They first created a dataset of 74 open-source projects and then used an existing code smell detection tool as an advisor to obtain a candidate list of data samples. Then, they manually validated 1,986 of these candidates as the training dataset. Based on this dataset, they employed 16 different machine learning algorithms to detect 4 types of code smells, and they horizontally compared the results with different algorithms. For large classes, the naïve Bayes algorithm exhibited the best performance.

Another machine-learning-based approach uses a clustering algorithm to identify extracted class opportunities. Classes with extracted class opportunities are identified as large class. JDeodorant [10][11] is a well-known code smell refactoring tool that applies the HAC algorithm to identify extracted class opportunities. This tool computes the entity sets for each class entity and compares the similarities between them using the Jaccard distance. Akash [12] also provided a HAC-based approach by computing the similarity matrix of the methods. The similarity matrix exploits both structural and semantic similarities between the methods using three cohesion metrics: structural similarity between methods (SSM), call-based dependence between methods (CDM), and conceptual similarity between methods (CSM).

Although the above studies proved that machine learning algorithms could be a valuable approach by which to identify large classes, achieving the best feature selection and threshold design remains a challenge. Furthermore, although the advisor-based dataset generation approach used by Fontana [7] reduces the workload of the human labeling task, finding smelly classes from a large-scale code corpus remains a labor-intensive task.

Deep learning, an important subfield of machine learning, has recently received extensive attention from researchers. A paper presented by Liu [8] described the construction of a neural network that consists of long short-term memory (LSTM) and dense layers to identify large classes. To obtain substantial data samples for deep learning tasks, they introduced an automatic dataset generation approach called smell-introducing refactoring, which involves unwanted refactoring that reduces the software quality. For positive data samples, they selected several high-quality open-source projects as the data sources and iterated all classes to find the class pairs that could be merged. The merged class was used directly as a positive sample. In contrast, the original classes in data source were assumed to be negative samples. Based on an automatically generated dataset, they used 12 types of software metrics and semantic information (methods and fields) as input data to train the classifier for large class detection.

To the best of our knowledge, this is the first use of software metrics with deep learning, and it opens a new perspective for code smell detection. However, this approach has certain limitations. First, as discussed in the paper, the quality of the automatically generated dataset cannot be guaranteed because it is difficult to ensure that all classes in open-source projects are well-designed, and they cannot guarantee that all merged class has the characteristics of a large class. Second, the network used in their approach was simple, and the input data relied heavily on software metrics.

According to the results of the aforementioned studies, in this study, we further extend our prior research [29], in which a GCN was utilized for long method detection, by applying it to the task of large class detection to address new challenges in the area of large class detection. First, we redesign the automatic generation techniques and grouping rules of the semiautomatic approach from our previous study [29] to generate sufficient class samples. Next, we define a new type of DHG as an input graph based on the methods similarity matrix and software metrics. Then, we approach the large class detection task as a graph classification problem to obtain a classifier for identifying large classes. We experiment with three GNN models and horizontally compare their results.

3. Methodology

3.1 Overview

Figure 1 illustrates the proposed approach. In Step.1, we redesigned the semiautomatic approach from our previous study [29] to generate a large number of data samples. In Step.2, we create a new type of DHG as the input graph for each class sample based on the methods similarity matrix. We calculate 22 types of software metrics (11 class-level and 11 method-level metrics) as graph node features. In Step.3, we take the large class detection task as the graph classification task. All graphs were input into the GNN model to train the classifier for large class detection.

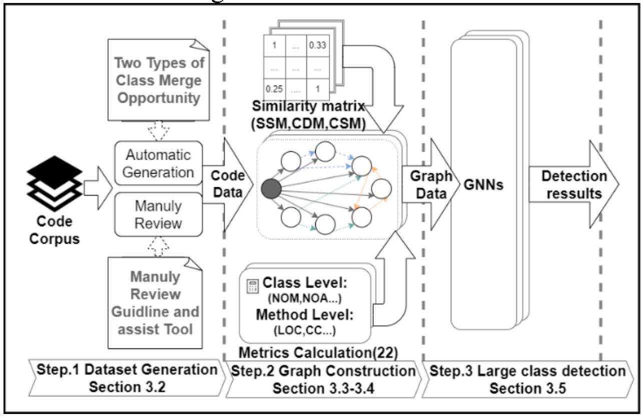

Two Types of Class Merge Opportunity
Automatic Generation
Manuly Review
Manuly Review Guidline and assist Tool
Code Corpus
Code Data
1 ... 0.33
0.25 1
Similarity matrix (SSM,CDM,CSM)
Class Level: (NOM,NOA...)
Method Level: (LOC,CC...)
Metrics Calculation(22)
Graph Data
GNNs
Detection ressults
Step.1 Dataset Generation Section 3.2
Step.2 Graph Construction Section 3.3-3.4
Step.3 Large class detection Section 3.5

Fig 1 Overview of proposed approach

3.2 Dataset Generation

In the first step, we redesigned the semiautomatic dataset generation approach based on the basic ideas of our previous study [29]. Specifically, we first defined two types of class-merging opportunities that could be automatically identified to generate a positive sample. Second, we utilized three class-level metrics to create the possibility range and defined the corresponding rules for grouping the data samples. Moreover, we developed a new assistive labeling tool to further accelerate the manual checking phase. The details of this approach are as follows.

To efficiently obtain sufficient high-quality data samples, it is necessary to reduce the workload of human labeling as much as possible. Thus, we must determine which classes have a higher possibility (or lower possibility) of being large class, so that we can focus human effort on the ambiguous data samples. To achieve this, we used three software metrics: lines of code (LOC), NOM, and NOA, to create three possibility ranges (PRs), as shown in Table 1.

Table 1 The possibility-range description.

	Metrics	Description
PR1	LOC > $\text{MaxTv}_{\text{LOC}}$ && NOM > $\text{MaxTv}_{\text{NOM}}$ && NOA > $\text{MaxTv}_{\text{NOA}}$	The class has great possibility to be a large class.
PR2	LOC < $\text{MinTv}_{\text{LOC}}$ && NOM < $\text{MinTv}_{\text{NOM}}$ && NOA < $\text{MinTv}_{\text{NOA}}$	The class has less possibility to be a large class.
PR3	other	The class has possibility to be a large class.

In Table 1, we set two tolerance values for each metric: the maximum tolerable value (MaxTv) and the minimum tolerable value (MinTv). When all metrics exceed their MaxTv, we consider that the class has a high possibility of being large (PR1). However, if all the metrics are below MinTv, we consider the class to have a lower possibility of being a large class (PR2). If a class fails to satisfy any of these conditions, it is considered to have the general possibility of being a large class (PR3). The MaxTv and MinTv values depend on the target program language and the projects used in the code corpus. We need to investigate the specifications of the target program language or obtain statistics on the above metrics in the code corpus before setting the value. In this study, we performed experiments using the Java language, and the setting of its value is explained in Section 4.1-B.

The main purpose of the above PRs is to assist in identifying data samples that require human verification, but not to directly make the identification of large class. In the following steps, we will divide the data samples into two groups (A_Group and M_Group) by following the rules listed in Table 2. The data samples in the A_Group will be directly applied to the final dataset, whereas the data samples in the M_Group are included after manual confirmation. With used this technique, we can focus more on those ambiguous data samples, thus reducing the workload of dataset production. After the PRs were established, the dataset was generated following the process illustrated in Fig 2.

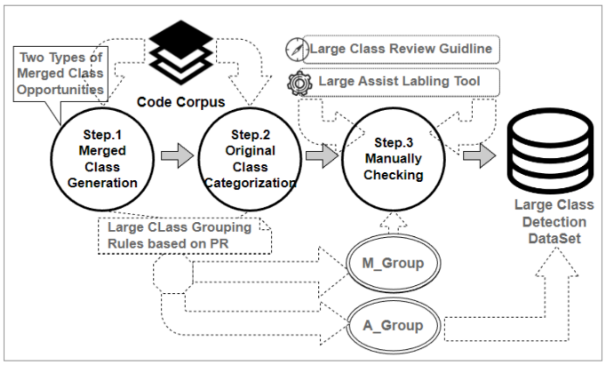

Two Types of Merged Class Opportunities
Code Corpus
Large Class Review Guidline
Large Assist Labling Tool
Step.1 Merged Class Generation
Step.2 Original Class Categorization
Step.3 Manually Checking
Large Class Detection DataSet
Large CLass Grouping Rules based on PR
M_Group
A_Group

Fig 2 Overview of dataset generation

Step.1 Merged Class Generation. In the first step of dataset generation, we focused on generating positive data samples (smelly classes) and categorizing them based on open-source projects in the code corpus. To achieve this, we first design two patterns of merging opportunities that can be automatically identified based on the three refactoring strategies for large class that were introduced by Fowler [2]: extract class, extract superclass, and replace type code with subclasses. These two types of merging opportunities are explained as follows.

The first pattern involves classes with an inheritance relationship, in which the parent class can be merged with the child class. For example, in pattern 1 of Fig 3, the parent class “Product” can be merged into the child class “Book” by copying the methods and fields to the child class. On the other hand, the second pattern encompasses a pair of classes with a usage relationship that can be combined. As illustrated in pattern 2 of Fig 3, the class “Cart” is used as a field in the “User” class. Thus, we can merge two classes by copying all fields and methods from the “Cart” class to the “User” class.

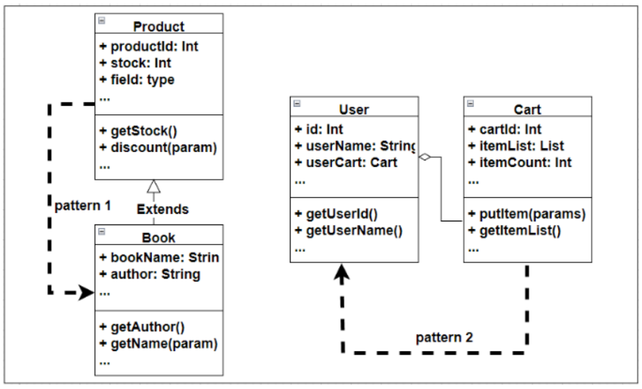


Fig 3 Two types of merge opportunities

Once the merge opportunities were identified, we merged the classes by copying methods and fields from the source class to the target class. Different merging opportunity patterns require different merging operations. For example, in the first pattern, only fields and methods that do not exist in the child class need to be copied. However, in the second pattern, all methods and fields must be copied to the target class. In addition, we must delete the target fields in the original class and change all the references. Note that not all merge opportunities could be successfully executed owing to potential merge failures, such as method name conflicts and multiple inheritances. Any merge opportunity that fails to be executed is removed from consideration. The number of failed merge opportunity may vary depending on the target project. In our experiments, we used a total of eight projects as shown in Section 4.1-B, and the average percentage of failed merge opportunity was about 40%.

After the merged classes were generated, we divided them into two groups (A_Group and M_Group) by following the rules listed in Table 2. The merged class with a possibility range in PR1 is placed in A_Group. The merged class with a possibility range in PR3 is placed into the M_Group.

Table 2 The data samples grouping rules.

Group	Rules
M_group	1.The merged class in PR3 2.The original class in PR3 3.The original class in PR1
A_Group	1.The merged class in PR1 2.The original class in PR2

Step.2 Original Class Categorization. The second step in the dataset generation is to iterate all classes in the code corpus and categorize them into the above two groups by following the rules in Table 2. If the original class is classified as PR2, we assume that the class is smell-free and put it into the A_Group as a negative data sample. However, if the original class is in PR1 or PR3, it is placed into the M_Group to wait for a human check.

Using this grouping technique, we did not need to check for classes that were less likely to be large. This reduces the human labeling cost of dataset generation. However, we could not guarantee extremely high quality of the negative samples because a large class may also exist in PR2. Nevertheless, considering both the cost and quality of the dataset, we still think this is an efficient approach. Furthermore, because we used the PRs to categorize the classes, only those classes that have a small chance of being large are automatically placed into the final dataset. Hence, the quality of our dataset can be greatly improved compared with the existing automatic dataset generation approach [8].

Step.3 Manually Checking. The last step in the dataset generation is labeling the code sample in the M_Group. A common problem in the manual checking phase is that different labeling results might be given for some ambiguous data samples owing to the different cognition of large class or the experience of the reviewers regarding the target project. To alleviate this issue, we set up a list of guideline questions based on the relevant prior studies [2][5] et al. The specific guidelines are as follows:

1. Does the class have too many lines of code?
2. Does the class have too many fields?
3. Does the class have too many complex methods?
4. Does the class have class extraction opportunities that may reduce the reusability of the target class?
5. Does the class have too many responsibilities, which may reduce the maintainability of the target class?

In this study, to further accelerate the manual checking phase, we developed a new assist tool to help developers quickly grasp the basic characteristics of the target class and make precise judgments. The tool is available at https://github.com/Bankzhy/lclb.

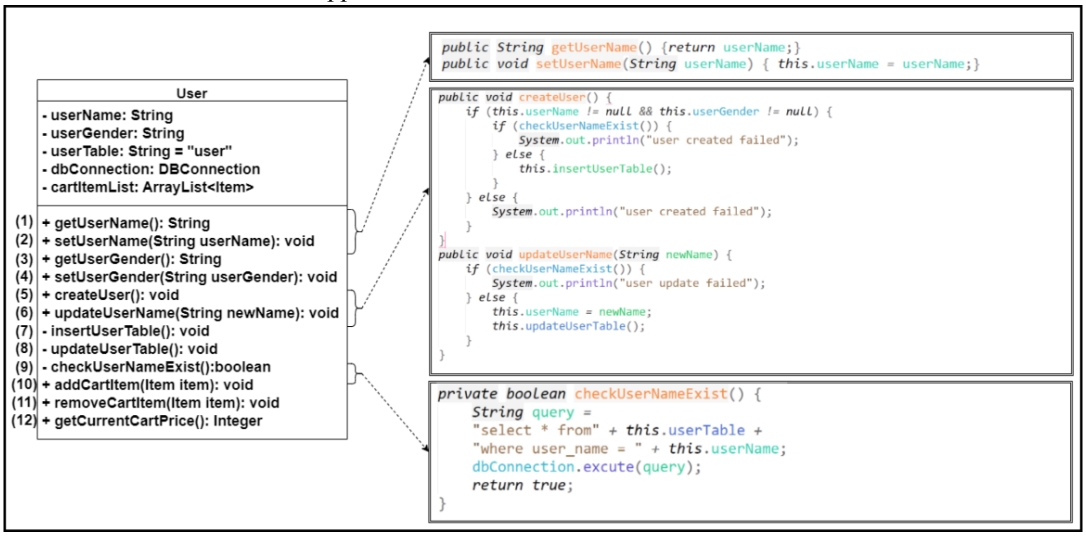
After manual confirmation, we need to merge the data samples from A_Group and M_Group. It's essential to consider the balance of data samples. In the experiments of this study, we will try to keep the ratio of positive to negative samples as 1:1. Additionally, to ensure the quality of the dataset, we set the proportion of automatically generated data samples in the final dataset should not exceed sixty percent.

3.3 Metrics Calculation

Software metrics are some of the most commonly used techniques in software quality assessment. In this study, we also calculated 22 types of software metrics from the method and class levels as node features of the input graph. At the method level, we primarily followed metrics from previous studies [29][13] that are appropriate for measuring methods, and at the class level, we referred to metrics from existing studies [6][14] that are often used to capture class characteristics. The details of the metrics used are as follows.

At the method level, we calculated the following 11 metrics. LOC, McCabe's cyclomatic complexity (CC), parameter count (PC), and nest block depth (NBD) are the four most commonly used metrics. The lack of cohesion method (LCOM1 to LCOM3) are three metrics introduced by Charalampidou [13], and the number of accessed variables (NOAV) is a metric introduced by Lanza [6].

The field-used count (FUC), local method using count (LMUC), and text similarity method class (TSMC) are three custom metrics. The FUC and LMUC represent the total


User
- userName: String
- userGender: String
- userTable: String = "user"
- dbConnection: DBConnection
- cartItemList: ArrayList<Item>
(1) + getUserName(): String
(2) + setUserName(String userName): void
(3) + getUserGender(): String
(4) + setUserGender(String userGender): void
(5) + createUser(): void
(6) + updateUserName(String newName): void
(7) - insertUserTable(): void
(8) - updateUserTable(): void
(9) - checkUserNameExist():boolean
(10) + addCartItem(Item item): void
(11) + removeCartItem(Item item): void
(12) + getCurrentCartPrice(): Integer

```
public String getUserName() {return userName;}
public void setUserName(String userName) { this.userName = userName;}
```

```
public void createUser() {
    if (this.userName != null && this.userGender != null) {
        if (checkUserNameExist()) {
            System.out.println("user created failed");
        } else {
            this.insertUserTable();
        }
    } else {
        System.out.println("user created failed");
    }
}
public void updateUserName(String newName) {
    if (checkUserNameExist()) {
        System.out.println("user update failed");
    } else {
        this.userName = newName;
        this.updateUserTable();
    }
}
```

```
private boolean checkUserNameExist() {
    String query =
    "select * from" + this.userTable +
    "where user_name = " + this.userName;
    dbConnection.excute(query);
    return true;
}
```

Fig 4 Example of graph construction - example class

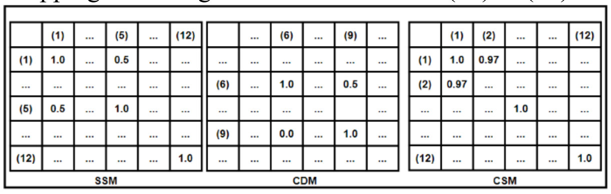
numbers of fields and methods in the target class, respectively. Furthermore, we used Word2Vec [21][22] to calculate the text similarity between the method name and class name for the TSMC.

At the class level, the following 11 metrics were calculated. The NOM, NOA, number of public attributes (NOPA), ATFD, WMC, and TCC are the metrics from Lanza [6]. The class interface size (CIS), direct class coupling (DCC), and cohesion among methods of class (CAM) are the metrics from Bansiya [14]. The depth of inheritance tree (DIT) and lack of cohesion in methods (LCOM) are the metrics from Chidamber [15].

3.4 Graph Construction

For the task with used GNNs, it is crucial to construct the input graph. In contrast to the method-level input graph based on the program dependency graph (PDG) proposed in our previous study [29], we define a new type of class-level DHG as an input graph based on previous studies [12][16].

To simplify the input graph construction process, we first introduce an example class, as shown in Fig 4. In this example, the “User” class comprises 5 fields and 12 methods, clearly demonstrating the presence of three distinct types of functions that are aggregated. The first type of function pertains to the operations performed by the “User” class, with the relevant methods (1) to (6). The second type of function involves the operations related to the “User” database, corresponding to methods (7) to (9). Finally, the third type of function pertains to the operations performed on the user shopping cart using the relevant methods (10) to (12).



	(1)	...	(5)	...	(12)
(1)	1.0	...	0.5	...	...
...	...	...	...	...	...
(5)	0.5	...	1.0	...	...
...	...	...	...	...	...
(12)	...	...	...	...	1.0

SSM

	...	(6)	...	(9)	...
...	...	...	...	...	...
(6)	...	1.0	...	0.5	...
...	...	...	...		...
(9)	...	0.0	...	1.0	...
...	...	...	...	...	...

CDM

	(1)	(2)	...	...	(12)
(1)	1.0	0.97	...	...	...
(2)	0.97	...	...	...	...
...	...	...	1.0	...	...
...	...	...	...	...	...
(12)	...	...	...	...	1.0

CSM

Fig 5 Example of graph construction - similarity matrix

To construct the input graph for the example class above, we first need to calculate the methods similarity matrix proposed by Akash [12], as shown in Fig 5. In this figure, we compute three metrics, SSM, CDM, and CSM, for each method to construct its corresponding similarity matrix. The details of the calculation are explained in the following paragraphs. Next, we built an input graph, as shown in Fig 6. To maintain the readability of the input graph, we split it into different edge types. The input graph comprises two types of nodes and four types of edges. The two types of nodes include the class node and method node, which represent the target class and all methods within it, respectively. For the edges of the input graph, we first created three types of edges between the methods: SSM, CDM, and CSM edges based on the above similarity matrix. Moreover, we created the include edge from the class node to all method nodes within

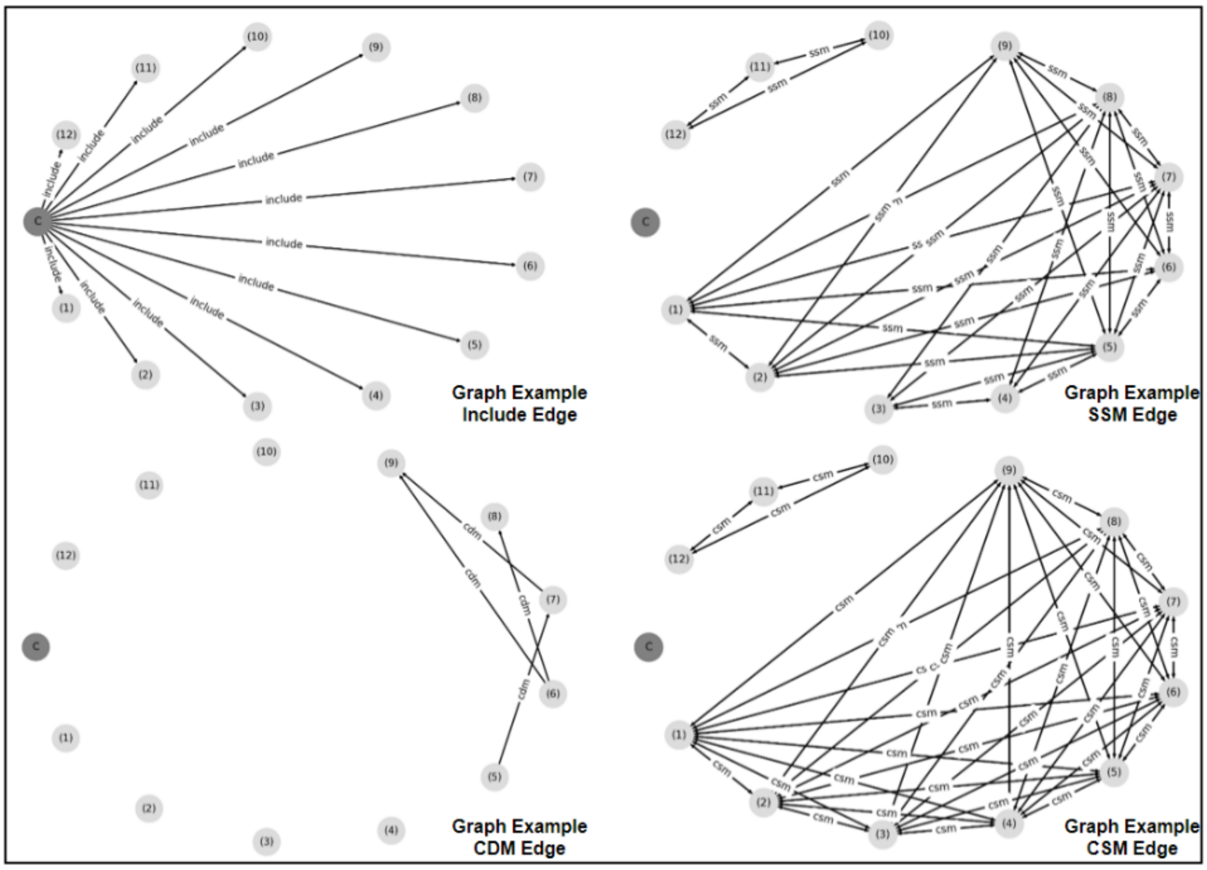


Fig 6 Example of graph construction - input graph

the target class. The details of the input graph are as follows.
Class Node: The class node represents the target class. Only one class node exists in the input graph. For example, in Fig 6, the class node "C" is the unique class node representing the target class "User" from Fig 4. In addition, the class-level metrics calculated in Section 3.3 are used as class node features.
Method Node: The method nodes represent methods within the target class. In the example shown in Fig 4, methods (1)–(12) are represented as nodes (1)–(12) in Fig 6. In addition, the method level metrics calculated in Section 3.3 are used as the method node features.
Include Edge: The edges from class node to method nodes. For example, in Fig 6, the class node "C" and all method nodes are connected by include edges.
SSM Edge: The SSM metric is the structural similarity proposed by Gui [17]. This metric is calculated based on the cohesion and transitivity between the methods. The SSM calculation is shown in Eq.2.

$$SSM_{i,j} = \begin{cases} \dfrac{|V_i \cap V_j|}{|V_i \cup V_j|} & if |V_i \cup V_j| \neq 0 \\ 0 & otherwise \end{cases} \quad (2)$$

In this formula, suppose that we have two methods: m_i and m_j. Then, V_i and V_j are the instance variables accessed by m_i and m_j. A higher SSM value suggests that m_i and m_j belongs to the same class.

In our approach, we calculated the SSM for each pair of methods. If the SSM value was greater than zero, we connected the two methods with a bi-directional SSM edge. For the example in Fig 4, we calculated the similarity matrix for all methods, as shown in Fig 5. Focusing on methods (1) and (5), we observed that $SSM_{1\text{-}5}$ exceeded the predefined threshold of 0 by 0.5. Consequently, we connected the two method nodes using an SSM edge, as shown in Fig 6.
CDM Edge: In contrast to SSM, CDM is another structural similarity metric calculated by method calls between methods. It was proposed by Bavota in 2011 [16], and its calculation is shown in Eq.3. In this formula, *calls*(m_i, m_j) denotes the number of times m_j is called from m_i, and $calls_{in}(m_j)$ is the total number of incoming calls for m_j.

$$CDM_{i\to j} = \begin{cases} \dfrac{calls(m_i, m_j)}{calls_{in}(m_j)} & if\ calls_{in}(m_j) \neq 0 \\ 0 & otherwise \end{cases} \quad (3)$$

In our approach, we calculate the CDM for each pair of methods. If the CDM value is greater than zero, we connect the two methods with a directional CDM edge from the caller method to the callee method. In the example class shown in Fig 4, the CDM between methods (6) and (9) exceeds the threshold of 0 by 0.5, as shown in Fig 5. In this case, the two method nodes are connected using a CDM edge, as shown in Fig 6.
CSM Edge: CSM represents the conceptual cohesion measure between each pair of methods within a class, as proposed by Poshyvanyk [18]. This measures how semantically the two methods are related. The calculation of this formula is shown in Eq.4. In this formula, the vector v is calculated via latent semantic indexing (LSI) [19]. However, the approach proposed by Akash [12] used the latent Dirichlet allocation [20] algorithm. In our approach, we use Word2Vec [21][22] to represent semantic information as a vector.

$$CSM_{i,j} = \frac{v_i^T \cdot v_j}{||v_i|| \cdot ||v_j||} \quad (4)$$

We calculated the CSM for each pair of methods. If the CSM value is greater than 0.5, the two method nodes are connected by a CSM edge. For example, the CSM between methods (1) and (2) was 0.97, exceeding the threshold of 0.5, as shown in Fig 5. In this case, we can connect the two method nodes using a CSM edge, as shown in Fig 6.

Note that in the above construction process, we set the connection thresholds for the SSM, CDM, and CSM edges, which are designed with the basic principle of making the input graph more informative. For SSM and CDM edge, they represent the structural information (common instance variables or calling relationships) among methods within the target class. Since the structural relationships are not commonly exist between methods, we set the thresholds for these two edges to zero. For instance, our experiments revealed that over 80% of inter-methods lacked these structural relationships in the code corpus of Section 4.1-B. However, for CSM edge, because some of the same common token often appears between methods (such as: "create", "get", etc.), setting the threshold to zero may not appropriate. Our investigation into the code corpus indicated that the average CSM value across different projects ranged from 0.5 to 0.7. Thus, based on the above design principle, we set the threshold value to 0.5 for the CSM edge.

3.5 GNNs for Large Class Detection

Graph neural networks (GNNs) are neural models that can be directly applied to graphs to capture interdependencies between nodes through message passing [23]. This was first mentioned by Gori [24] in 2005 and further elaborated by Scarselli [25] in 2009. GNNs have made significant progress and are used in various areas, including social network analysis, protein interface prediction, and recommendation systems. To date, various model architectures of GNNs have been proposed, and we applied three widely used GNN model architectures: GCN [27], GraphSage [26], and GAT [28], to the large class detection task. Each of these three model architectures has unique characteristics. In brief, GCN is a basic GNN model that obtains node feature information from itself and all neighboring nodes. GraphSage acquires node representation by aggregating information from neighbor sampling. GAT uses an attention mechanism to learn the weights from different nodes. A detailed description and comparison of the three model architectures are provided in [30].

First, we constructed an input graph for each data sample according to the graph construction approach described in Section 3.4. Next, we considered the identification of a large class as a graph classification task. Then, we applied the above three GNN models to construct the network and fed all the input graphs to train the classifier for large class detection. A detailed experiment and performance evaluation are described in Section 4.

4. Evaluation Experiment

4.1 Experiment Design

A. Research Questions

In the evaluation experiments, we aim to verify the effectiveness of the proposed approach and determine whether its performance is superior to that of existing approaches. Moreover, because there are different ratios of large classes in real projects, we further validate the effectiveness of the proposal approach under different ratios of large classes. We mainly focus on the following research questions.

Q1. Does the proposed large class detection approach yield satisfactory training results when using the training dataset, and can the resulting classifier be used to detect large classes?

Q2. How does the performance of the proposed approach compare with those of existing large class detection approaches?

Q3. How does the proposed approach perform for different ratios of positive data samples?

B. Training Dataset

In our experiment, we first collected the training dataset using the process described in Section 3.2.

First, we collected code corpus from eight popular open-source projects: Junit4 [33], Mybatis3 [34], RxJava [40], JEdit [35], Netty [36], PMD [37], Gephi [38], and Libgdx [39]. These projects are well-known and come from a variety of usage areas. Next, we set up the possibility ranges as described in Section 3.2. The values of MaxTv and MinTv for each metric were determined based on the recommendations proposed by Lanza [6], who suggested that the average LOC was 70 in Java programming and that when the LOC exceeded 130, it appeared to be high. In terms of NOM, the average value could be 7, and the highest value could be 10 in Java programming. However, we could not find an explicit standard for the value of NOA. Thus, we examined the aforementioned eight projects and observed that the average NOA ranged from 0 to 5, whereas for classes with LOC values exceeding 130, the average of NOA fell within the range of 5–10. For this reason, the MinTv and MaxTv of each metric were set as shown in Table 3.

Table 3 The value of MaxTV and MinTv.

Group	NOA	NOM	LOC
MaxTv	10	10	130
MinTv	5	7	70

After preparation, we implemented the assistance program to process the dataset generation. The program first transforms all target projects to the abstract syntax tree (AST) using the “tree-sitter” [41]. The program then generates the merged class and divides it into two groups, following the rules in Section 3.2-Step.1. Next, the program iterates all the original classes and categorizes them according to the rules in Section 3.2-Step.2. Finally, with the help of three Java developers, we performed manual checking in the M_Group based on the guidelines and assistance program introduced in Section 3.2-Step.3. Of the three Java developers, two were master students, and one was an assistant teacher. All three developers came from the Department of Computer Science and have extensive learning experience in both software design and the Java language. Moreover, two of them had more than one year of work experience. Furthermore, we conducted tutorials in large class for all developers and distributed guidebooks before the experiment. With the help of the assistance program developed in Section 3.2-Step.3, all three developers collaborated remotely to perform the labeling process, and the results were stored directly in a centralized database. Although we were unable to accurately measure the overall time cost, it took approximately 1–5 minutes for each class sample.

Using this process, in total, we obtained 4,510 merged class samples and approximately 10,000 original class samples. After grouping and manual checking, we constructed a dataset of 3,102 positive data samples and 3,495 negative data samples. Approximately 30% of the positive and 50% of the negative samples were manually checked.

C. Evaluation Dataset

To prove the validity of the proposed approach, an independent evaluation dataset was created. In contrast to the training dataset, the evaluation dataset was created based on five open-source projects: OpenRefine [42], Jgrapht [43], Freeplane [44], Open Hospital [45], and Jsprit [46], and it was completely reviewed by above three Java developers.

We assigned three Java developers to manually validate the classes in the source projects by following the guidelines in Section 3.2-Step.3. Once the number of reviewed positive sample classes reached approximately 200, manual confirmation ceased, and an equal number of corresponding negative samples were selected to construct the dataset.

D. GNN Model Architectures and Configurations

In our experiment, the training networks of all three GNN models were constructed according to the architecture in Fig 7. The network consists of two GNN layers and one linear layer. In addition, Adam was used as the optimizer, and cross-entropy was used as the loss function. The experiments relied primarily on the PyTorch [47], whereas the implementation of the GNN predominantly utilized DGL [48].

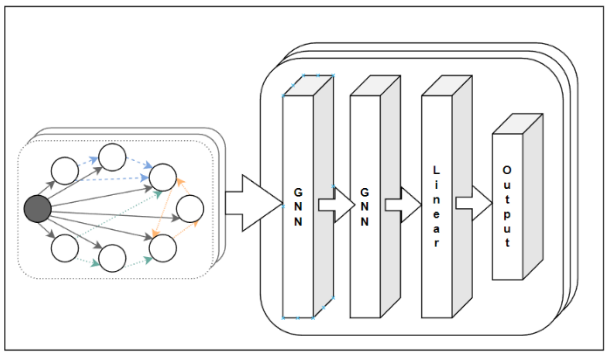


Fig 5 The model architecture of GNN models

E. Evaluation Criteria

In this study, we employed three commonly used metrics to evaluate the detection performance of the proposed approach: precision, recall, and F1. Precision is the ratio of positive samples marked by developers to detected positive samples. Recall is the probability of being predicted as a positive sample for all classes labeled as positive by developers. F1 is the harmonized average of the above two metrics, and it is calculated using Eq.5.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5}$$

4.2 Experiment Results

Q1. Does the proposed large class detection approach yield satisfactory training results with used the training dataset, and if the result classifier could be used to detect large classes?

For the first research question, we applied the training dataset to the above GNN. We randomly selected 80% of the training dataset to train the network and the other 20% as the test dataset to observe the training results. To make the results more plausible, we repeated the process five times and performed 5-fold cross-validation. Table 4 presents the averages of the results.

Table 4 The training results of large class detection.

Metrics	GCN		GraphSage		GAT	
	POS	NEG	**POS**	NEG	**POS**	NEG
Precision	**0.89**	0.88	**0.98**	0.98	**0.97**	0.97
Recall	**0.86**	0.91	**0.97**	0.98	**0.97**	0.98
F1	**0.88**	0.90	**0.98**	0.98	**0.97**	0.98

The results in Table 4 show that from an overall perspective, although all three GNN models achieved good training results, the results of the GCN (0.88) were lower than those of the other models. In positive sample detection, GraphSage and GAT did not show a significant performance gap, with GraphSage having a slightly higher F1 score than GAT (0.98 and 0.97).

Because the differences in Table 4 were small, we further tested the differences for each GNN model using McNemar's test [31], which is a statistical test that can be used to compare two classifiers in machine learning [32]. We employed all five classifiers from each GNN model to make predictions and determined a positive classification result only when over 80% of the classifiers predicted a positive result. The results show that GAT and GraphSage do not show significant differences when the significance level is 0.05; however, GCN shows significant differences from both GAT and GraphSage.

From the above results, we can make the following observations. First, all three GNN models achieved good training results, and the classifiers can be applied to large class detection. Second, the training results of the GAT and GraphSage were significantly higher than those of the GCN. However, GraphSage and GAT did not exhibit a significant performance gap, with the mean of GraphSage being slightly higher than that of GAT.

Q2. How does the performance of proposed approach compare to the existing large class detection approach?

To answer this question, we compared the detection performance of our approach with those of the existing clustering-based approach proposed by Akash [12] and the deep learning approach proposed by Liu [8]. We used the dataset from Section 4.1-B as the training dataset, and the dataset in Section 4.1-C as the evaluation dataset. The evaluation results are shown in Fig 8.

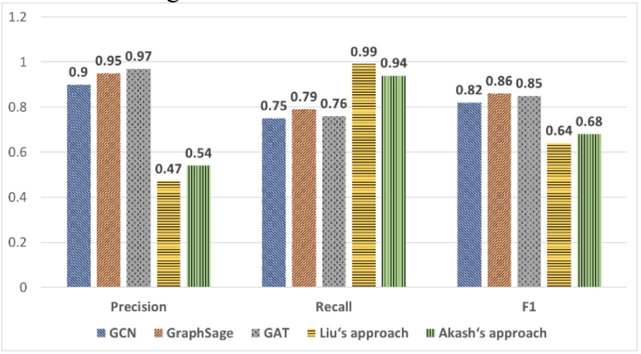


Fig 6 The detection results compared with those of existing approaches

From the results in Fig 8, we could make following observations.

- The F1-score of each GNN model exceeded 0.80, which is higher than those achieved by the existing approaches proposed by Akash [12] (0.68) and Liu [8] (0.64). Overall, GraphSage achieved the best detection performance (0.86).
- The recall of the existing approaches [8][12] have both reached above 0.90 higher than all GNN-based approaches proposed by us. This means that the existing deep learning approach [8] and clustering-based approach [12] have greater abilities to capture large classes from code samples.
- However, the precision of all GNN models in our approach exceeded 0.90, which is significantly higher than those of the existing approaches [8][12] (0.47 and 0.54), indicating that the existing approaches [8][12] may detect many erroneous samples. Conversely, our approach could identify the large class more accurate.

Q3. How does the proposed approach perform under different ratios of positive data samples?

For the assessment of Q2, we used an evaluation dataset with positive samples accounting for 50%. However, in practical software projects, the proportion of large classes may differ according to the project quality. Thus, we adjusted the proportion of positive samples in the dataset by increasing the number of negative samples in the evaluation dataset to test the detection performance under different values (10–60%). The results of F1-score are shown in Fig 9. From the results, we could make following observations:

- Although the GCN had a slightly lower F1-score than the other models, GraphSage and GAT achieved nearly the same value. The results of GraphSage were slightly better than those of the other models.
- From the comparison results of our approach with the existing clustering-based approach proposed by Akash [12], we observed that the F1 of the existing clustering-based approach [12] increases significantly as the proportion of positive samples increases. When the proportion of positive samples is less than 30%, the existing clustering-based approach [12] does not perform well. The main reason for this result is that the clustering-based approach aims to find the extract class opportunities from the target class. However, despite being identified as having extraction opportunities, some classes still do not meet the criteria for being marked as large by reviewers in terms of the number of methods or complexity of the class. Therefore, when the proportion of positive samples is small, the precision of this approach is significantly reduced. However, compared with the significant shifts in the existing study [12], our approach has showed more stability and better detection performance for any positive sample ratio.
- Compared with the existing deep learning approach [8], our approach exhibited a more stable detection performance. The proposed approach outperforms the existing deep learning approach [8] when the proportion of positive samples ranges from 10% to 50%. However, when the proportion of positive samples reaches 60%, the numerical growth of the existing deep learning approach [8] quickly surpasses that of the proposed approach.

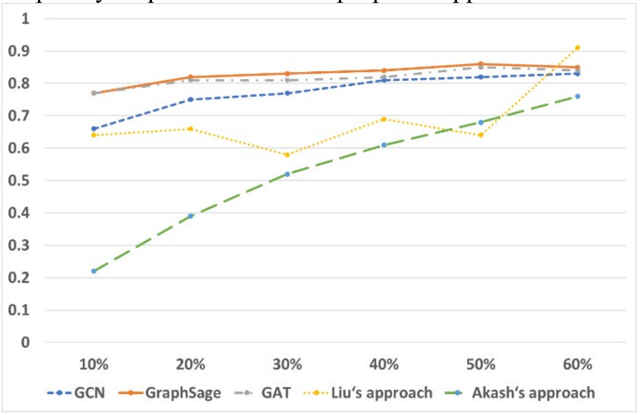


Fig 7 The F1-score under different ratios of positive data samples

We can summarize the results as follows. First, among all experimental GNN models, GraphSage performed slightly better. Second, our approach shows better accuracy than do those from prior studies [8][12]. Meanwhile, our approach showed a more stable detection performance for different ratios of positive data samples.

5. Conclusion

In this study, we applied an important division of deep learning: GNNs, to large class detection. To obtain sufficient data samples for the deep learning task, we redesigned the automatic generation techniques and grouping rules of the semi-automatic approach in our previous study [29], which could generate a large number of data samples with less human labeling effort. Using this generation approach, we obtained a total of 6,597 data samples for large class detection.

After generating the dataset, we designed a new type of DHG based on the methods similarity matrix and software metrics. We treated the identification of large classes as a graph classification task and fed all the input graphs to train the classifier for large class detection.

In the evaluation experiment, three typical GNNs were applied for large class detection. By using the trained classifiers, we compared the detection performance of the proposed approach with the existing clustering-based approach proposed by Akash [12] and the deep learning approach proposed by Liu [8]. The results showed that the proposed approach achieved better accuracy and more stable detection performance under different ratios of positive data samples.

In future work, we intend to extend our study in the following aspects. First, we shall extend our approach to multiple classification tasks in the future, as a large class could be further classified into several levels, depending on the severity of the situation. Further, we can apply the GNN-based approach to other code smells, such as feature envy and duplicated code.

References

[1] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis, University of Illinois at Urbana, 1992.

[2] M. Fowler, ed., Refactoring: improving the design of existing code, Addison-Wesley Signature, 1999 1st ed, 2018 2nd ed.

[3] T. Sharma, D. Spinellis, "A survey on software smells," Journal of Systems and Software, vol.138, pp.158-173, 2018.

[4] A. AbuHassan, M. Alshayeb, L. Ghouti, "Software smell detection techniques: A systematic literature review," Journal of Software: Evolution and Process, vol.33, e2320, 2021.

[5] W.J. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray, ed., Anti patterns: refactoring software, architectures, and projects in crisis, John Wiley and Sons, 1998.

[6] M. Lanza, R. Marinescu, ed., Object-Oriented Metrics in Practice, 2006.

[7] F.A. Fontana, M.V. Mäntylä, M. Zanoni, A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empirical Software Engineering, vol.21, pp.1143-1191, 2016.

[8] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, L. Zhang, "Deep Learning Based Code Smell Detection," IEEE Transactions on Software Engineering, vol.47, pp.1811-1837, 2021.

[9] N. Moha, Y. Gueheneuc, L. Duchien, A.L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," IEEE Transactions on Software Engineering, vol.36, pp.20-36, 2009.
[10] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," ICSE '11: Proceedings of the 33rd International Conference on Software Engineering, pp.1037-1039, 2011.
[11] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, " Identification and application of Extract Class refactorings in object-oriented systems," Journal of Systems and Software, vol.85, pp.2241-2260, 2012.
[12] P.S. Akash, A.Z. Sadiq and A. Kabir, "An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity," ENASE 2019, pp.427-433, 2019.
[13] S. Charalampidou, A. Ampatzoglou, P. Avgeriou, "Size and cohesion metrics as indicators of the long," PROMISE '15, vol.8, pp.1-10, 2015.
[14] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering, vol.28, pp.4-17, 2002.
[15] S.R. Chidamber; C.F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol.20, pp.476-493, 1994.
[16] G. Bavota, A.D. Lucia, A. Marcus, R. Oliveto, "A two-step technique for extract class refactoring," ASE '10, vol.20, pp.151-154, 2010.
[17] G. Gui, P.D. Scott, "Coupling and cohesion measures for evaluation of component reusability," MSR '06, pp.18-21, 2006.
[18] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," Empirical Software Engineering, vol.14, pp.5-32, 2009.
[19] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, Richard Harshman, "Indexing by latent semantic analysis," Journal of the American Society for Information, vol.41, pp.391-407, 1990.
[20] D.M. Blei, A.Y. Ng, M.I. Jordan, " Latent dirichlet allocation," Journal of Machine Learning Research, vol.3, pp.993-1022, 2003.
[21] T. Mikolov, K. Chen, G. Corrado, J. Dean, "Efficient Estimation of Word Representations in Vector Space," Journal of Machine Learning Research, arXiv:1301.3781 [cs.CL].
[22] T. Mikolov, K. Chen, G. Corrado, J. Dean, "Distributed representations of words and phrases and their compositionality," Proceedings of the 26th International Conference on Neural Information Processing Systems, vol.2, pp.3111-3119, 2013.
[23] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, "Graph neural networks: A review of methods and applications," AI Open, vol.1, pp.57-81, 2020.
[24] M. Gori, G. Monfardini and F. Scarselli, "IEEE International Joint Conference on Neural Networks," 2005.
[25] F. Scarselli, M. Gori and A.C. Tsoi, "The graph neural network model," IEEE Transactions on Neural Networks, vol.20, pp.61-80, 2009.
[26] W.L. Hamilton, R. Ying, J. Leskovec, "Inductive representation learning on large graphs," NIPS'17, pp.1024-1034, 2017.
[27] T.N. Kipf, M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," ICLR'2017, 2017.
[28] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, "Graph Attention Networks," ICLR 2018, 2018.
[29] HY Zhang, TJ Kishi, "Long Method Detection using Graph Convolutional Networks," Journal of Information Process, vol.31, pp.469-477, 2023.
[30] B. Khemani, S. Patil, K. Kotecha, S. Tanwar, "A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions," Journal of Big Data, 18, 2024.
[31] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," Psychometrika, vol.12, pp.153-157, 1947.
[32] S Raschka, "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning," arXiv:1811.12808 [cs.LG].
[33] Junit, https://github.com/junit-team/junit4. accessed Nov.29.2022.
[34] Mybatis, https://github.com/mybatis/mybatis-3. accessed Nov.29.2022.
[35] JEdit, http://www.jedit.org/index.php?page=download, accessed Nov.29.2022.
[36] Netty, https://netty.io/, accessed Nov.29.2022.
[37] PMD, https://github.com/pmd/pmd, accessed Nov.29.2022.
[38] Gephi, https://gephi.org/, accessed Nov.29.2022.
[39] Libgdx, https://libgdx.com/, accessed Nov.29.2022.
[40] RxJava. https://github.com/ReactiveX/RxJava, accessed Nov.29.2022.
[41] tree-sitter. https://github.com/tree-sitter/, accessed Nov.29.2022
[42] OpenRefine. https://github.com/OpenRefine/, accessed Nov.29.2022
[43] Jgrapht. https://jgrapht.org/, accessed Nov.29.2022
[44] Freeplane. https://github.com/freeplane/, accessed Nov.29.2022
[45] Open Hospital. https://github.com/informatici, accessed Nov.29.2022
[46] Jsprit. https://github.com/graphhopper/jsprit, accessed Nov.29.2022
[47] PyTorch. https://pytorch.org/, accessed Nov.29.2022
[48] DGL. https://www.dgl.ai/, accessed Nov.29.2022

Hanyu Zhang received M.S. in Deparment of Industrial and Management Systems Engineering, Waseda University. He is currently working at Inner Mongolia University of Science & Technology.

Tomoji Kishi received M.S. in Information Science, Graduate School of Engineering, Kyoto University. Ph.D. from Japan Advanced Institute of Science and Technology (JAIST). After working at NEC and JAIST, he has been a professor at the Department of Industrial and Management Systems Engineering, Waseda University.