IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 1, 2021, accepted February 6, 2021, date of publication February 10, 2021, date of current version February 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3058450

Mining Key Classes in Java Projects by Examining a Very Small Number of Classes: A Complex Network-Based Approach

HAO LI[1], TIAN WANG[1], WEIFENG PAN[1], MUCHOU WANG[2], CHUNLAI CHAI[1], PENGYU CHEN[3], JIALE WANG[1], AND JING WANG[4]

[1]School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China
[2]Wenzhou University Library, Wenzhou University, Wenzhou 325000, China
[3]Zhuoyue Honors College, Hangzhou Dianzi University, Hangzhou 310018, China
[4]School of Software and Communication Engineering, Jiangxi University of Finance and Economics, Nanchang 330013, China

Corresponding authors: Tian Wang (wtaddiction@163.com), Weifeng Pan (wfpan@zjgsu.edu.cn), and Muchou Wang (wzuwmc@163.com)

This work was supported in part by the Natural Science Foundation of Zhejiang Province under Grant LY21F020002, in part by the National Key Research and Development Program of China under Grant 2017YFB1400602, and in part by the Key Research and Development Program Project of Zhejiang Province under Grant 2019C01004.

ABSTRACT Key classes have become excellent starting points for developers to understand unknown software systems. Up to now, a variety of approaches have been proposed to mine key classes in a software project. Many of them are based on a network representation (namely, software networks) of the software projects. However, the software networks they used are usually un-weighted and un-directed, which is not consistent with the reality in a real software project where the coupling actually has direction and strength. Worse still, the number of key class candidates returned by existing approaches is usually very large. Thus, it is usually infeasible for developers to start the comprehension process from these classes, especially when there are tight time and resource constraints. To tackle these problems, in this paper, we propose an approach named MinClass, to Mine key Classes in Java projects by examining a very small number of classes. First, the software structure at the class level is represented by a weighted directed software network, which considers both the coupling strength and direction between every pair of classes. Second, we propose a new metric, OSE (One-order Structural Entropy), and use it to calculate the importance of each class in the system. Finally, we sort classes in descending order according to their OSE values, and a small number of top-ranked classes are treated as the key class candidates identified by MinClass. Experiments are performed on six open-source Java projects, and comparison studies with other eight state-of-the-art approaches are also performed. Results show that, although no one method performs best in all software systems, MinClass is the most promising one. It performs best in the whole set of software systems according to the average ranking of the Friedman test. Thus, MinClass is a valuable technique that can be used to mine the key classes.

INDEX TERMS Complex network, key classes, static analysis, program comprehensions.

I. INTRODUCTION

Software is always born to solve real problems, but real problems and requirements are always changing [1]. Therefore, software needs continuous adjustments and evolution to be adapted to changes in the real world. Understanding a software project is usually the first step for software engineers to adjust and modify it. We usually call this behavior *software maintenance*, which has been the most costly part of the software life cycle [2]. If developers can quickly understand the unknown software project, then they can reduce the cost of software maintenance timely. Obviously, understanding the core components of a software project is a good entry point to help developers quickly familiarize themselves with unknown systems [3].

Before the rise of object-oriented (OO for short) software projects, developers always understood software projects from the perspective of modules. When OO software projects

The associate editor coordinating the review of this manuscript and approving it for publication was Engang Tian.

TABLE 1. Properties of the subject software system.

System	Version	Directory	*LOC*	*#packages*	*#classes* (*#enums*)	*#methods*	*#attributes*	*URLs*
Ant	1.6.1	main	81515	67	900	7691	4167	https://github.com/apache/ant
GWTPortlets	0.9.5beta	src	8501	10	145	1145	424	https://code.google.com/archive/p/gwtportlets/
jEdit	5.1.0	src	112492	41	1082(9)	7601	4085	http://www.jedit.org/index.php
JHotDraw	6.0b.1	src	28330	30	544	5205	865	http://sourceforge.net/projects/jhotdraw
maze	1	src	8881	6	63(6)	563	284	http://code.google.com/p/maze-solver/
Wro4J	1.6.3	src	33736	30	567(9)	3256	1274	https://github.com/wro4j/wro4j

became mainstream, developers usually first understood the key classes of software projects [4]. Software documentation can provide new developers with relevant information about key classes. However, in practice, most software development teams do not provide complete and detailed documentation. Therefore, we need to obtain the key classes of a software project through other approaches instead of relying on documents [3].

Complex networks provide a new perspective for identifying key classes of software projects [5]–[7]. The identification of key nodes is an important topic in the field of complex network research [8]. When a software project is referred to as *software network* (i.e., nodes represent classes (interfaces), and edges represent relationships between classes (interfaces)) [9]–[11], we can use techniques in the complex networks to identify key classes in software projects, and many interesting results have been reported [3]–[18]. However, there are many shortcomings in the literature. For one thing, the software networks they used are usually un-weighted and un-directed, which is not consistent with the reality in a real software project where the coupling actually has direction and strength. For another, the existing work usually used a threshold 15% to mine the key classes, i.e., the top-ranked 15% classes are treated as key classes. However, even 15% is used, the number of key class candidates returned by existing approaches is still very large. For example, even for a small software project Ant (see Table 1) with only 900 classes, if a threshold 15% is applied, the existing approaches will recommend 135 classes as key classes. Thus, it is still infeasible for developers to start the comprehension process from the 135 classes, especially when there are tight time and resource constraints.

The aim of this paper is to apply the key node identification technology in complex networks to software networks, so as to provide developers with an ordered list of a small number of key classes (we use term *classes* to denote both *classes* and *interfaces* from here on) for an unknown software project. To this aim, we propose an approach, named MinClass, to Mine key Classes in Java projects by examining a very small number of classes. First, the software structure at the class level is represented by a weighted directed software network, which considers both the coupling strength and direction between every pair of classes. Second, we propose a new metric, *OSE* (One-order Structural Entropy), and use it to calculate the importance of each class in the system. Finally, we sort classes in descending order according to their *OSE* values, and a small number of top-ranked classes are treated as the key class candidates identified by MinClass. Experiments are performed on six open-source Java software projects, and comparison studies with eight other state-of-the-art approaches are also performed. Results show that, although no one method performs best in all software projects, MinClass is the most promising one. It performs best in the whole set of software projects according to the average ranking of the Friedman test.

The contribution of this work can be summarized as follows:

- We propose a new problem on key class identification, i.e., identifying key classes by only examining a very small of classes in a specific software project.
- We propose a new metric *OSE* to measure the importance of classes in the software network. It is a metric which is based on both the weight on the link and the direction of the link. Our metric characterizes much more information of the software structure.
- We provide a replication package for the experiments performed in our empirical studies. Interested readers can use the data set and tool therein to replicate our research (see https://github.com/SEGroupZJGSU/MinClass).

The rest of this paper is structured as follows: Section II briefly summarizes the related work on identifying key classes in software projects, and Section III introduces our software network and *OSE* metric in detail. Section IV provides the experimental validation of our approach, with focus on the answering of our raised research questions. In Section V, we give the conclusion remark and discuss the future work.

II. RELATED WORK

Many results on key class identification have been reported in the last few years. These studies can be roughly classified into two groups, i.e., approaches based on dynamic analysis and approaches based on static analysis. For the approaches based on dynamic analysis, they heavily depend on the dynamic execution scenarios of the target Java project.

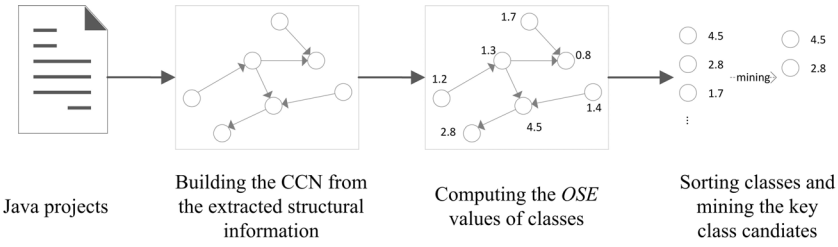


FIGURE 1. The framework of our MinClass approach.

Thus, to perform the task, it is the first step to obtain complete execution scenarios, which is usually infeasible in practice, especially for large-scale software projects in the sense that to obtain all possible execution scenarios in an acceptable time period seems impossible. For the approaches based on static analysis, the data set required for a specific approach is easy to be collected. Thus, compared with the approaches based on dynamic analysis, approaches based on static analysis have great potential to be applied in practice. In this work, we mainly discuss the studies which are based on static analysis.

Zaidman *et al.* [3] used the HITS (Hypertext Induced Topic Selection) webmining algorithm to mine key classes from a software project. Perin *et al.* [5] modeled the software structure as a directed graph, and then leveraged the PageRank algorithm to mine key classes. Zhou *et al.* [6] modeled the software structure as a dependency graph, and leveraged the *h*-index and its invariants to mine key classes. Steidl *et al.* [7] modeled the software structure of a software project as a dependency graph, and characterized the properties of classes using some metrics such as *betweenness*, *PageRank*, and *HITS* so as to mine key classes. Osman *et al.* [12] leveraged machine learning algorithms to mine key classes by compressing the class diagram of the target software project. Thung *et al.* [13] improved the performance of Osman *et al.*'s approach by combining design metrics and network metrics together in the learning processes. Meyer *et al.* [14] modeled software projects as software networks and leveraged *k*-core decomposition to mine key classes. Jiang *et al.* [15] measured the importance of classes using finite state machine approach. Şora and Chirila [16] compared the performance of different approaches such as *PageRank*, *HITS* and *Degree* in the key-classes identification problem, and recommended to use network metrics to mine key classes in practice. Pan *et al.* [17] represented software projects as software networks at class level, and proposed a generalized k-core decomposition algorithm to compute the coreness of each class in the software network, which is further used to mine the key classes. Vale and Maia [18] proposed a semi-automatic way, Keecle, to collect key classes in a target software project; it is based on a dynamic analysis of the subject system.

III. THE MINCLASS APPROACH

The framework of our MinClass approach is shown in Figure 1. The MinClass approach mainly includes three parts: (1) software structural information extraction: analyzing the software source code to obtain structural information; (2) software network definition: defining a weighted directed software network and using it to formally represent the structural information; and (3) *OSE* metric definition: defining an *OSE* metric and using it to measure the importance of each class in a software system. In the following subsections we will elaborate on the main parts of our approach.

A. SOFTWARE STRUCTURAL INFORMATION EXTRACTION

MinClass also follows the line of approaches based on static analysis to mine key classes from a target software project. Thus, it also relies on the structural information enclosed in the source code of the target project. To extract the structural information and further represent it as a software network, we refer to our own-developed software analysis platform SNAP (Software Network Analysis Platform) [17]. SNAP can extract software structural information, and also contains many network analysis and visualization functions. In this section, we use SNAP to extract various software elements (e.g., *classes*, *interfaces*, *attributes*, *methods*, and *local variables*) and their coupling relationships (e.g., *inheritance* relation, *implementation* relation, and *method call* relation) in the target software project. Note that, in this work, we only consider the software elements which are actually defined in the target project, i.e., software elements which are only referenced are ignored.

B. SOFTWARE NETWORK DEFINITION

After extracting the structural information of a target project, we use a weighted directed network CCN (Class Coupling network) to formally describe it. In the CCN, we did not distinguish classes from interfaces, and regarded them as the same kind of elements, i.e., the term *class* (or *classes*) designates both the classes and interfaces. When a class uses the services provided by another class, a directed link will be established in the CCN to represent the coupling between the two classes. The weight on the link represents the coupling strength between the two classes. Thus, CCN can be formally defined as

$$\mathrm{CCN} = (N, L),$$

where N is the node set, each of which denotes a class in the target projects; L is the link set, each of which denotes the coupling between a specific pair of classes. Note that, in the CCN, we consider a total of seven types of

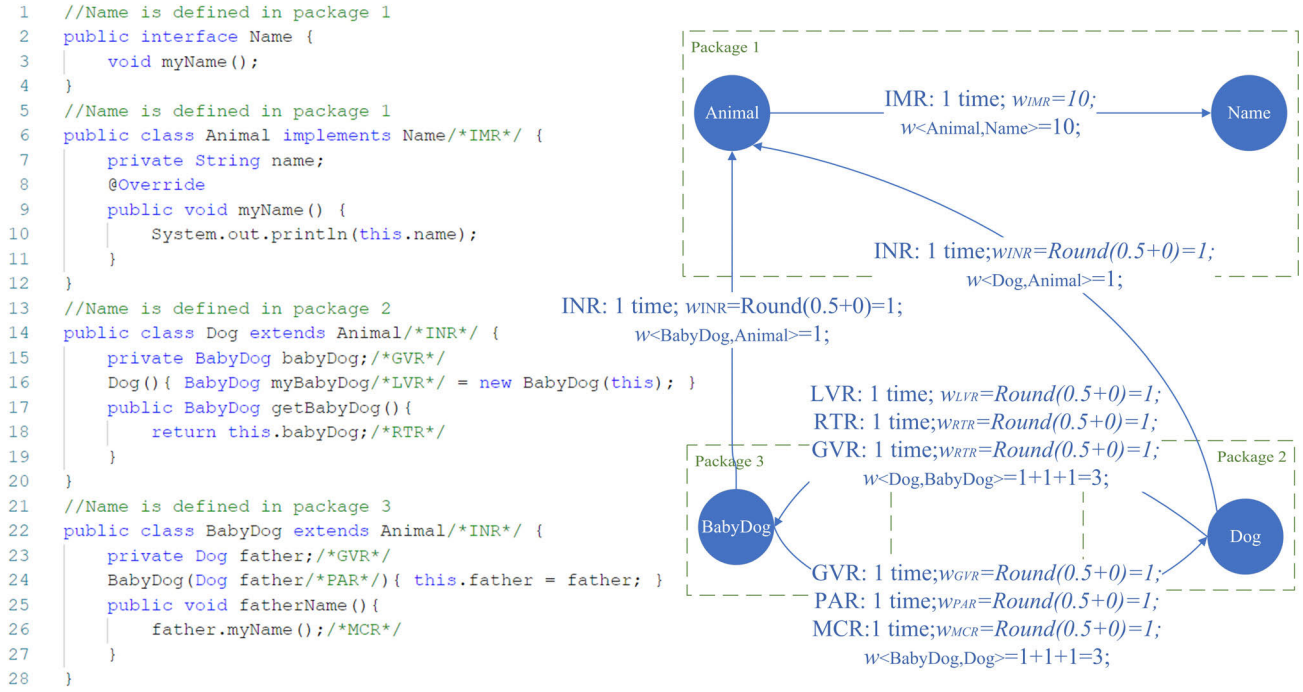


FIGURE 2. A simple code snippet (the left part) and its corresponding CCN (the right part).

couplings between classes, i.e., *inheritance* relation (*INR*), *implements* relation (*IMR*), *parameter* relation (*PAR*), *global variable* relation (*GVR*), *method call* relation (*MCR*), *local variable* relation (*LVR*), and *return type* relation (*RTR*). These couplings are very common in Java projects.

We use a simple example shown in the left part of Figure 2 to illustrate the seven types of couplings that we consider in this work. As shown in Figure 2, there is an *INR* coupling between classes *Dog* and *Animal* since *Dog* extends *Animal* (see line 14); there is an *IMR* coupling between class *Animal* and interface *Name* since *Animal* implements *Name* (see line 6); there is a *PAR* coupling between classes *BabyDog* and *Dog* since the constructor of *BabyDog* has a parameter with type of *Dog* (see line 24); there is a *GVR* coupling between classes *BabyDog* and *Dog* since *BabyDog* has an attribute with type of *Dog* (see line 23); there is a *MCR* coupling between classes *BabyDog* and *Dog* since the method *faterName* of *BabyDog* calls the method *myName* of Dog *father* (see line 26); there is a *LVR* coupling between classes *Dog* and *BabyDog* since a local variable *myBabyDog* of type *BabyDog* is defined in *Dog* (see line 16). These coupling types are mapped to different links in the corresponding CCN (see the right part of Figure 2).

In the CCN, we assign weights to the links between every pair of classes if they interact with each other. These weights are used to represent the coupling strength between the corresponding class pairs. Obviously, two classes may interact with each other via multiple coupling types (see the couplings between classes *Dog* and *BabyDog*), and different types of couplings may differ in strength. Thus, how to compute the weights to express the strength of multiple couplings between classes is a question worth exploring.

Abreu *et al.* [19], [20] proposed an object approach to quantify the strength of different coupling types by tracing the intra- and inter-module distribution of different coupling types in the whole software project. Specifically, the coupling strength for different coupling types is computed as follows:

$$w_r = \begin{cases} 10, & N^r_{\text{intra}} \neq 0 \wedge N^r_{\text{inter}} = 0 \\ 1, & N^r_{\text{intra}} = 0 \wedge N^r_{\text{inter}} = 0 \\ Round(0.5 + 10 \times \frac{N^r_{\text{intra}}}{N^r_{\text{intra}} + N^r_{\text{inter}}}), & otherwise, \end{cases} \tag{1}$$

where $r \in \{INR, IMR, PAR, GVR, MCR, LVR, RTR\}$ is a specific coupling type; w_r is the strength assigned to the coupling of type r; N^r_{intra} and N^r_{inter} are the count of intra- and inter-module coupling, respectively. *Round(x)* is the rounded integer of x.

As mentioned above, there might exist more than one coupling types between a specific pair of classes. For example, in Figure 2, classes *Dog* and *BabyDog* have a total of three coupling types, i.e., *GVR*, *PAR* and *MCR*. Furthermore, different coupling types may have different coupling frequencies. Thus, in this work, the final weight on each link is computed as follows:

$$w_{ij} = \sum\nolimits_r f^r_{ij} \times w_r, \tag{2}$$

where f^r_{ij} is the count of the coupling of type r on the link $< c_i, c_j >$ ($r \in \{INR, IMR, PAR, GVR, MCR, LVR, RTR\}$), and

w_r is the strength of the coupling of type r, which is computed according to their intra- and inter-module distribution. For example, there is only one instance of *IMR* between class *Animal* and interface *Name*. Thus, according to formula (1), w_{IMR} equals to 10, and the final weight on the link equals to 10 (see the right part of Figure 2).

C. OSE

As mentioned above, in the CCN of a target software project, the weight on the link is used to measure the coupling strength between two classes, and the direction is used to represent the coupling direction. Thus, as an effective metric to measure the importance of classes in software systems, it should consider both the coupling strength and coupling direction between classes. Specifically, when designing a metric to measure class importance, we should take the following characteristics into consideration:

- *The difference in the coupling strength between class nodes in the CCN*: the coupling strength between nodes is neglected in the traditional un-weighted networks, and is regarded as the same. Weighted networks use weights on the links to quantify the coupling strength between classes. Generally, the weights on links are often not the same, which will affect the relative importance of different classes.
- *The difference in the number of neighbors of different class nodes in the CCN*: different nodes often have different numbers of neighbors, and such a difference may affect the importance of different classes.
- *The difference in the distribution of weights on the links between class nodes and their neighbors*: different classes often have different neighbor sets, with the weight sets on all the links between these class nodes and their neighbors also being different. Thus, the weight-distributions of class nodes and their neighbors are different, which might greatly affect the importance of class nodes.
- *The difference in the importance of neighbors of class nodes*: different class nodes have different neighbors, and the importance of neighbors is usually different. Neighbors with difference importance will make different contributions to the importance of class nodes. Thus, we should consider the difference in the contributions that a specific neighbor makes to the class nodes.

In this work, we propose a new metric named *OSE* to measure the importance of classes. *OSE* can be formally defined as:

$$P_{xw} = \frac{w_{xw}/w_{\max}}{\sum_{y\in in(w)} w_{yw}/w_{\max}}, x, \quad w \in N, \tag{3}$$

$$h_w = \left(1 - \sum_{x\in in(w)} (P_{xw} \ln P_{xw})\right) \sum_{x\in in(w)} w_{xw}/w_{\max}, \quad w \in N, \tag{4}$$

$$OSE_w = h_w + \sum_{v\in in(w)} \left(\frac{w_{vw}/w_{\max}}{\sum_{u\in on(v)} w_{vu}/w_{\max}} h_v\right), \quad w \in N, \tag{5}$$

where N is the class node set in the CCN; x and w are two class nodes in N; $in(w)$ is the in-neighbors of class w; $on(v)$ is the out-neighbors of class v; w_{xw} is the weight on the link $< x, w >$; OSE_w is the *OSE* of class w; $w_{\max}$is the maximum weight on the links in the CCN.

Note that our *OSE* metric takes all the three characteristics that we have mentioned above into consideration. Specifically, MinClass considers the difference in the coupling strength between classes by containing w_{xw} in formula (5), considers the difference in the number of neighbors by containing $in(w)$ and $on(v)$ in formulae (3), (4) and (5), considers the difference in the distribution of weights on the links by containing $-\sum_{x\in in(w)} (P_{xw} \ln P_{xw})$ in formula (4), and considers the difference in the importance of neighbors by considering

$$\sum_{v\in in(w)} \left(\frac{w_{vw}/w_{\max}}{\sum_{u\in on(v)} w_{vu}/w_{\max}} h_v\right)$$

in formula (5).

Note that, for some special classes like utility classes, they might be used by a lot of other classes and thus might be identified by *OSE* as key classes, even if these classes do not contain much functionality. As mentioned above, our approach ignores the referenced software elements. Thus, the referenced simple utility classes will be ignored. But for the utility classes defined in the target project, even if they are very simple, they might still be identified as key classes, thus affecting the performance of our approach. *OSE* cannot differentiate these simple utility classes from the classes in the target project and filter them out. Generally, filtering out these simple utility classes can improve the performance of a specific approach, but it is not the focus of this work.

Figure 3 give a simple example to show our process to compute the value of *OSE*. We take node B as an example. First, we compute the value of $h_w(w \in \{A, B, C, D, E, F\})$ of each node according to formula (3) and formula (4), and the results are shown in the middle part of Figure 3. After all the results are obtained, the results are substituted into the formula (5) (see the right part of Figure 3) to compute OSE_B, which is the value of *OSE* for class B.

D. CLASS IMPORTANCE SORTING

After obtaining the *OSE* values of all the classes in the target software project, we sort all the classes in the CCN in descending order. Then a threshold k is used to mine the key classes, i.e., classes with a rank less than k (i.e., top-k) will be regarded as the key class candidates returned by MinClass.

IV. EMPIRICAL STUDY

In this section, we perform a set of experiments on six Java projects to validate the effectiveness of our MinClass approach.

A. SUBJECT SOFTWARE PROJECTS

We use a total of six Java projects (see Table 1) as our subject systems to validate our MissClass approach. The six

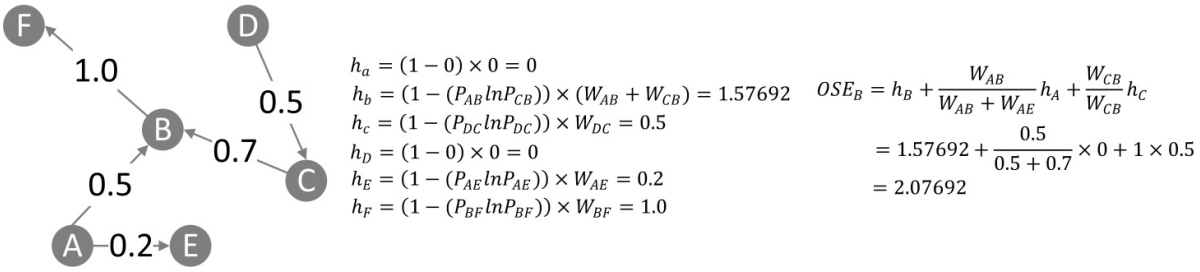


FIGURE 3. Illustration of the process to compute *OSE*.

TABLE 2. The number of different coupling types and the strength of each coupling type using the form as *number of intra-couplings |number of inter-couplings|strength of the coupling type*.

System	*INR*	*IMR*	*PAR*	*GVR*	*MCR*	*LVR*	*RTR*
Ant	263|254|6	104|54|7	440|928|4	122|300|3	5610|4580|6	485|990|4	238|944|3
GWTPortlets	29|42|5	9|32|3	75|153|4	50|56|5	726|510|6	41|126|3	50|106|4
jEdit	130|117|6	97|69|6	886|879|6	274|336|5	11069|5933|7	949|1007|5	579|224|8
JHotDraw	67|183|3	34|67|4	200|828|2	45|312|2	3005|2534|6	106|450|2	113|568|2
maze	19|3|9	1|4|3	25|65|3	33|26|6	703|423|7	63|78|5	16|14|6
Wro4J	77|62|6	53|139|3	49|274|2	145|269|4	2669|1750|7	119|521|2	38|262|2

Java projects are regarded as the benchmark systems in the literature, which are widely used to validate the effectiveness of any approach on key class mining. The source code of these projects is downloaded from their homepages.

In Table 1, column *System* shows the name of the subject Java projects, column *Version* shows the version of the corresponding software project that we analyzed, column *LOC* is the line of code in the system, and columns *# packages, # classes(# enums), # methods*, and *# attributes* denote the number of packages, classes (enums), methods, attributes in the subject system, respectively. Column *URLs* shows the URL used to download the corresponding software system. Note that *LOC* does not contain the number of comments and blank lines; *# classes* contains the number of interfaces, inner classes and anonymous classes. Table 2 shows the number of intra-couplings, number of inter-couplings, and the strength of the corresponding coupling type. The three values are separated by "|".

B. RESEARCH QUESTIONS

To validate the effectiveness of our MinClass approach, we focus on the following two research questions (RQs):

- RQ1: Is our MinClass approach better than the existing approaches? There are many related approaches on mining key classes in software projects. Thus, we will perform a set of experiments to check whether our approach is better than the related approaches in the literature.
- RQ2: Does the MinClass approach have good scalability? As a feasible approach on mining key classes in software projects, MinClass should have the ability to be applied to software projects of different scale. Thus, we will perform a set of experiments to check whether our approach has the ability to be applied to software projects with a large number of classes.

C. EVALUATION METRICS

Any approaches on mining key classes organize the classes in a target software project into two groups, i.e., key classes and non-key classes. The key class mining problem is actually a binary classification problem. Thus, in this work, we use the two metrics, *precision* and *recall* for the comparison of different approaches.

For the key classes mining problem, its corresponding confusion matrix is shown in Figure 4, where *TP* denotes the number of key classes in the reference set which are also predicted by a specific approach as key classes, *FP* denotes the classes which are not in the reference set but are predicted by a specific approach as key classes, *FN* denotes the number of key classes in the reference set which are not predicted as key classes by a specific approach, and *TN* denotes the number of classes which are not in the reference set and are also predicted as non-key classes by a specific approach. Based on the definitions of *TP*, *FP*, *FN* and *TN*,

		Actual label	
		Key classes	Non-key classes
Predicted label	Key classes	*TP*	*FP*
	Non-key classes	*FN*	*TN*

FIGURE 4. The confusion matrix for the key classes mining approach.

TABLE 3. Comparison of the results obtained by the approaches applied to ant.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
IntrospectionHelper	×	×	×	×	×	×	×	×	×
Main	×	×	×	×	×	×	×	×	×
Project	√	√	√	×	√	√	√	√	√
ProjectHelper	×	×	×	×	×	×	×	×	×
RuntimeConfigurable	√	√	×	×	×	×	×	×	×
Target	√	√	×	×	×	√	×	×	×
Task	√	√	√	×	√	√	×	√	√
TaskContainer	×	×	×	×	×	×	×	×	×
UnknownElemen	√	√	×	×	×	×	×	×	×
ElementHandler	×	×	×	×	×	×	×	×	×
Recall (%)	50	50	20	0	20	30	10	20	20
Precision (%)	20	20	8	0	8	12	4	8	8

the evaluation metrics, *precision* and *recall* can be defined as [22]

$$precision = \frac{TP}{TP + FP}, \tag{6}$$

$$recall = \frac{TP}{TP + FN}, \tag{7}$$

Note that the reference set is composed of the true key classes in the target software project. In this work, the true key classes are retrieved from the design documents, where the original developers explicitly provided a short description to give an overview of the key classes of the target project in free texts or diagrams. Generally, the true key classes can be found in the sections of *architectural overview*, *core of the system*, and *introduction for developers* in the design documents.

D. BASELINE APPROACHES

As we have reviewed in the related work, there are many approaches on mining key classes in software projects. We select a total of eight approaches as our baseline approaches, i.e., PageRank [5], [7], [16], HITS [3], [7], Betweenness [7], Coreness [14], [21], InDeg (In-Degree) [16], OutDeg (Out-Degree) [16], Deg (Degree) [16], and ICOOK (Identifying key Class candidates in OO software using generalized K-core decomposition) [17]. We do not select other approaches as baline approaches mainly because we cannot replicate their approaches easily duing to the lack of sufficient information to implement their approaches. These approaches can be briefly described as follows:

- PageRank: PageRank has been widely used to measure the node importance in networks, and also has been used to measure class importance. In this work, we compute PageRank values of all class nodes in the CCN of the target software project.
- HITS: HITS (Hyperlink-Induced Topic Search) is an algorithm to rate Web pages by computing two scores (i.e., Hub and Authority) for each node in a network. In this work, we compute the Authority values of class nodes in the CCN as their importance.
- Coreness: the k-core is defined on a un-weighted network, which is the largest subgraph whose nodes with a degree $> k$. Then the coreness of a node is k if it belongs to k-core but does not belong to $(k + 1)$-core. The coreness has ever been used to measure class importance. In this work, we compute the coreness of class nodes in the CCN. Note that when computing the coreness, we do not consider the link weight and direction.
- Betweenness: the betweenness centrality of a node is the ratio of the number of shortest path containing the node to the number of shortest paths in the network. It has ever been used to measure the class importance. In this work, we compute the betweenness values of class nodes in the CCN as their importance.
- InDeg: InDeg denotes the in-degree of a node in a directed network, which quantifies the number of out-going links of a node.
- OutDeg: OutDeg denotes the out-degree of a node in a directed network, which quantifies the number of in-going links of a node.
- Deg: Deg denotes the degree of a node. In the CCN, it equals to the sum of InDeg and OutDeg, i.e., Deg = InDeg + OutDeg.
- ICOOK: ICOOK represented the target software project as a software network at the class level, and used a generalized k-core decomposition algorithm to compute the coreness of each class in the software network.

E. RESULTS AND ANALYSIS

In this work, we used our own-developed software analysis platform, SNAP, to analyze the subject software projects

TABLE 4. Comparison of the results obtained by the approaches applied to GWTPortlets.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
WidgetFactory	√	√	√	√	√	√	×	√	×
WidgetFactoryVisitor	√	√	×	×	×	×	×	×	×
WidgetRefreshHook	×	×	×	×	×	×	×	×	×
PageEditor	√	√	×	×	√	√	√	√	√
BroadcastListener	√	√	×	×	×	×	×	×	×
BroadcastManager	√	√	×	×	×	√	×	×	×
PageTitleChangeEvent	×	×	×	×	×	×	×	×	×
LDOM	√	√	√	×	√	√	√	√	√
Layout	√	√	√	√	√	√	×	×	√
LayoutConstraints	√	√	√	√	√	√	×	√	√
PositionAware	√	√	×	×	×	×	×	×	×
RowLayout	√	√	√	√	×	√	×	√	√
ClientAreaPanel	×	×	×	×	×	×	×	×	×
ContainerPortlet	×	×	×	√	×	×	√	√	√
LayoutPanel	√	√	√	√	√	√	√	√	√
PagePortlet	×	×	×	√	√	×	×	√	√
Portlet	√	√	√	√	√	√	√	√	√
ShadowPanel	×	×	×	×	×	×	×	√	×
Theme	√	√	×	×	×	√	×	√	√
TitlePanel	×	×	×	√	√	√	√	√	√
ToolButton	√	×	√	×	×	√	×	×	√
FormBuilder	√	×	×	×	×	√	×	×	×
Rectangle	√	√	√	×	√	√	×	√	√
PageProvider	×	×	×	×	√	×	×	×	×
PageRequest	×	√	×	×	√	×	×	×	×
WidgetDataProvider	×	×	×	×	×	×	×	×	×
WidgetFactoryXmlIO	×	×	×	×	×	×	×	×	×
Recall (%)	59.3	55.6	33.3	33.3	44.4	55.6	25.9	44.4	48.2
Precision (%)	64	60	36	36	48	60	28	48	52

shown in Table 1, i.e., extracting the structural information, building the CCN and applying MinClass to sort classes.

We provide a replication package for the experiments performed in our empirical studies. Interested readers can use the data set and tool (see https://github.com/SEGroupZJGSU/MinClass) therein to replicate our research.

In this section, we perform a set of experiments with the aim to answer our research questions.

1) RQ1: IS OUR MINCLASS APPROACH BETTER THAN THE EXISTING APPROACHES?

As we all know, the scale of software systems is increasingly large; software systems usually have thousands of classes and interfaces. Thus, even if a very small proportion of classes (e.g., treating the top 15% ranked classes as candidate key classes) are selected as candidate key classes; the number of candidate key classes is still very large, making the recommended classes cannot be used in practice, especially when the resources are limited. Şora and Chirila [16] considered that the number of key classes in a specific software system is not directly proportional to the size of software system, and the average number of key classes is between 20 and 30. Thus, in this work, we are limited to examine only the classes whose positions are less than 25 in the ranked list of classes, i.e., we only examine a very small number of classes to mine the key classes.

Tables 2 to 8 show the results of the nine approaches (i.e., our MinClass approach and the eight approaches in the baseline) on the six subject software systems. The first column *key classes* is the key classes in the reference set,

TABLE 5. Comparison of the results obtained by the approaches applied to jEdit.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
Buffer	√	√	√	×	√	√	√	√	√
EBMessage	×	√	×	×	×	×	×	×	×
EditPane	√	√	×	×	√	×	√	√	√
View	√	√	√	×	√	√	√	√	√
jEdit	√	√	√	×	√	√	√	√	√
JEditTextArea	×	×	×	×	√	×	×	×	√
Log	√	√	√	×	√	√	×	√	√
Recall (%)	71.4	85.7	57.1	0	85.7	57.14	57.14	71.43	85.7
Precision (%)	20	24	16	0	24	16	16	20	24

TABLE 6. Comparison of the results obtained by the approaches applied to JHotDraw.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
DrawApplication	√	×	√	√	√	√	√	√	√
Drawing	√	√	√	√	√	√	×	√	√
DrawingEditor	√	√	√	√	√	√	×	√	√
DrawingView	√	√	√	√	√	√	×	√	√
Figure	√	√	√	√	√	√	×	√	√
Handle	√	√	×	√	×	×	×	×	×
Tool	√	×	√	√	×	√	×	×	√
CompositeFigure	×	×	×	√	×	×	√	√	√
StandardDrawingView	×	×	×	√	√	×	√	√	√
Recall (%)	77.8	55.6	66.7	88.9	66.7	66.7	33.3	77.8	88.9
Precision (%)	28	20	24	32	24	24	12	28	32

and the other columns are the results of the corresponding approach, where "√" denotes the corresponding classes are predicted as key classes, and "×" denotes the corresponding approach misses the key classes. The last two rows show the *recall* and *precision* of the corresponding approaches, respectively. As mentioned above, we only consider the classes who rank in the top-25 of the ranked list of classes, i.e., the classes whose ranking positions before 25 are predicted as key classes, and those larger than 25 are predicted as non-key classes and will be filtered out.

From Table 3, we can see that there are ten key classes in the reference set of Ant; our MinClass and PageRank achieve a same level of performance with the *recall* and *precision* being 50% and 20%, respectively. They perform best among the nine approaches.

It can be seen from Table 4 that, when MinClass applied to GWTPortlets, it achieves a precision of 64%, and a recall of 59.3%, which are better than that of the approaches in the baseline.

As shown in Table 5, MinClass finds a total of five key classes in jEdit. Compared with PageRank, MinClass misses class *EBMessage*. By referring to the source code of this class, we found that *EBMessage* is an abstract class in the system. It is not used very frequently in jEdit; it cannot be predicted by MinClass as a key class. Compared with the results of Betweeness and ICOOK, the results of MinClass miss class *JEditTextArea*. By a close inspection of the CCN, we observed that the node representing *JEditTextArea* has a larger in-degree and smaller out-degree, which means *JEditTextArea* is used much more frequently by other classes when compared with its reliances on the services provided by other classes. *JEditTextArea* is more inclined to be a tool class. Our MinClass approach fails to recognize it as a key class.

In can be seen from Table 6 that, MinClass finds a total of seven key classes in JHotDraw, which is one less than that of coreness and ICOOK. However, for coreness, eight out of nine classes are found in the same core; their positions all equal to 25. In this sence, coreness is not a very effective approach since it cannot differentiate the classes in the same core. Although the number of key classes found by our approach is slightly less than coreness, our approach has a better ability to distinguish different key classes in the ranked list.

Table 7 shows the results obtained on the software system Maze. From Table 7, we can observe that our MinClass performs worse than Deg but no worse than other seven

TABLE 7. Comparison of the results obtained by the approaches applied to Maze.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
Main	√	√	×	√	√	√	×	√	√
Floodfill	×	×	√	×	×	×	√	√	√
LeftWallFollower	×	×	×	×	×	×	×	×	×
RightWallFollower	×	×	√	×	×	×	×	×	×
RobotBase	√	√	×	√	√	√	×	√	√
RobotController	√	×	√	√	√	√	√	√	√
RobotStep	×	×	×	×	×	×	×	×	×
Tremaux	×	×	√	×	×	×	√	√	√
MazeView	√	√	√	√	√	√	√	√	√
PrimaryFrame	√	√	×	√	√	√	√	√	√
BoxTemplate	×	×	×	×	×	×	×	×	×
CornerTemplate	×	×	×	×	×	×	×	×	×
Corner	×	×	×	×	×	×	×	×	×
CrossTemplate	×	×	×	×	×	×	×	×	×
EditableMazeView	×	×	√	×	√	×	√	√	√
MazeTemplate	√	√	√	×	√	√	×	√	√
StraightTemplate	×	×	×	×	×	×	×	×	×
TemplatePeg	√	√	×	×	√	√	×	√	×
TemplateWall	√	√	×	×	×	√	×	×	×
ZigZagTemplate	×	×	×	×	×	×	×	×	×
CellSizeModel	√	√	×	×	×	√	×	√	√
Direction	√	√	×	×	×	√	×	×	×
MazeCell	√	√	×	√	√	√	×	√	√
MazeModel	√	√	√	√	√	√	√	√	√
PegLocation	×	×	×	×	×	×	×	×	×
RobotModel	√	√	×	√	√	√	√	√	√
RobotModelMaster	√	√	√	√	√	√	√	√	√
Recall (%)	51.9	48.2	37.0	33.3	44.4	51.9	33.3	55.6	51.9
Precision (%)	56	52	40	36	48	56	36	60	56

approaches in the baseline. Specifically, Deg identifies a total of fifteen key classes in the reference set while MinClass identifies a total of fourteen key classes in the reference set.

Since the set of key classes found by the two is not highly consistent, we believe that these two approaches have different behaviors when mining key classes in a target software system. By a deep exploration, we observe that MinClass is easier to miss the key classes whose out-degree is much greater than the in-degree. However, for Deg, since it ignores the direction information of links, it is not affected by the direction information and performs better. On the contrary, for key classes whose in-degree is much greater than out-degree, our MinClass seems better than Deg.

Table 8 shows the results obtained on the software system Wro4J. From Table 8, we observe that MinClass performs no worse than other eight approaches in the baseline. Specifically, MinClass is better than PageRank, Deg, Betweenness, InDeg, and OutDeg, but achieves a same level performance with HITS and coreness.

The answer to RQ1: Obviously, in the above experiments, no one approach always performs best in all the cases, including the state-of-the-art approaches. Thus, we use the average ranking of the Friedman test to help us better compare the performance of different approaches in the whole set of subject systems. Table 9 shows the results of the average ranking of the Friedman test with regard to recall and precision. Note that, for the metrics recall and precision used in this work, a larger ranking value indicates a worse approach. Thus, the Friedman test sorts the nine approach into the following order: MinClass > ICOOK > PageRank > Deg > InDeg >

TABLE 8. Comparison of the results obtained by the approaches applied to Wro4J.

Key classes	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
WroFilter	×	×	√	√	√	√	√	√	√
WroManager	×	×	√	√	√	×	√	√	√
WroManagerFactory	√	×	√	√	×	×	×	×	×
WroModel	√	×	√	√	×	×	×	×	×
WroModelFactory	×	×	×	√	√	×	×	×	×
Group	√	×	√	√	×	√	×	√	√
Resource	√	√	√	√	√	√	×	√	√
ResourceType	√	√	×	×	×	×	×	×	×
UriLocator	√	√	√	×	√	√	×	×	×
UriLocatorFactory	√	√	√	√	√	×	×	×	√
ResourcePostProcessor	√	√	×	×	√	×	×	×	×
ResourcePreProcessor	√	√	√	√	√	√	×	√	×
Recall(%)	75	50	75	75	66.7	41.67	16.67	41.67	41.67
Precision(%)	36	24	36	36	32	20	8	20	20

TABLE 9. Results of the average ranking of the friedman test.

Approaches	MinClass	PageRank	HITS	Coreness	Betweenness	InDeg	OutDeg	Deg	ICOOK
Recall	2.50	4.00	5.58	6.17	4.92	4.58	8.42	4.42	4.42
Precision	2.58	4.08	5.58	6.25	5.00	4.75	8.42	4.50	3.83

Betweenness > HITS > Coreness > OutDeg, which means MinClass performs best and OutDeg performs worst in the whole set of subject systems.

2) RQ2: DOES THE MINCLASS APPROACH HAVE GOOD SCALABILITY?

As a feasible approach on mining key classes in software projects, MinClass should have the ability to be applied to software projects of different scale. To validate the scalability of MinClass, we traced the CPU time of our MinClass when applied to the subject software projects (see Table 10). Obviously, our MinClass is very efficient. The CPU time of MinClass on all the subject systems is all less than 1 minute. For example, jEdit has more than 1,000 classes, but the CUP time of MinClass on jEdit is only 48 seconds. Note that theCPU time shown in Table 10 is measured in seconds, and the time less than 1 second is omitted.

We also compare the CPU time of MinClass with that of the approaches in the baseline (see Figure 5). Obviously, the nine approaches are all very efficient. Even for the most inefficient approach Betweenness, its CPU time is still less than 1.5 minutes.

TABLE 10. The CPU time of MinClass on the subject software projects.

Software	Ant	GWTPortlets	jEdit	JhotDraw	Maze	Neuroph	Wro4j
Time(s)	33s	3s	48s	19s	1s	5s	14s

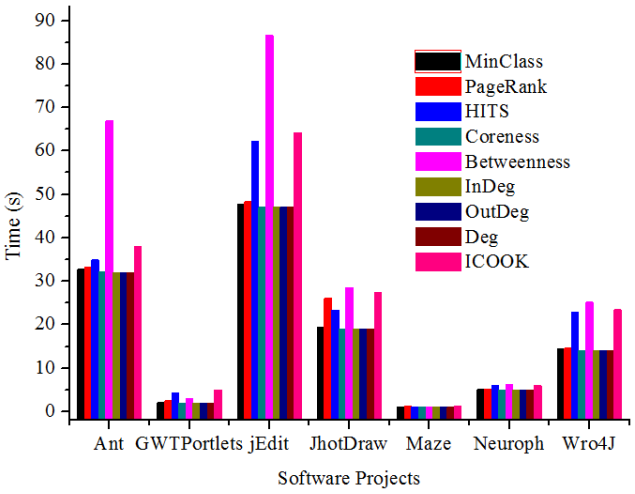


FIGURE 5. The CPU time of different approaches.

The answer to RQ2: From the results shown above, we can observe that our MinClass approach and the approaches in the baseline are all very efficient. Specifically, our MinClass can mine the key classes for all subject software projects in 1 minute; the approaches in the baseline can mine the

key classes for the subject software projects in 1.5 minutes. Our MinClass approach has the ability to be applied to larger software projects.

F. THREATS TO VALIDITY

The first threat to the effectiveness of our work mainly comes from the construction of the reference set of key classes. The reference set of key classes we used in this work is constructed based on the design document of the corresponding software system. In practice, when writing the documents, the document writer may be inclined to the part developed by the writer himself (or herself). Thus, the document might not be very objective. We partially mitigated this threat by using the benchmark software systems which are widely used in the literature.

The second threat comes from the subject software projects that we used in the experiments. Our subject systems are all open-source Java systems, which are different from the software systems developed using other programming languages such as Python, C++ and C. Thus, the conclusions obtained in Java software projects have the risk to be generalized to other non-Java software projects. In the future, we need to replicate this work on more non-Java software projects.

The third threat comes from the utility classes defined in the target software projects. Utility classes usually perform special functionality in the projects and may be heavily used by a lot of other classes. They might be identified by our *OSE* as key classes, even if these classes do not contain much functionality, thus affecting the performance of our approach. However, in the literature, to the best of our knowledge, no approach considers the special role that the utility classes play. Our approach also suffers from the losses of effectiveness result from these simple utility classes. We partially mitigated this threat by the objective weighting mechanism used to quantify the strength of different coupling types by tracing the intra- and inter-module distribution of different coupling types in the whole software project. For utility classes, they might have much more inter-module couplings than intra-module couplings, thus reducing the strength of this coupling type (see formula (1)). This might reduce the probability of utility classes to be identified as key classes. To further mitigate this threat, we need to identify these simple utility classes and filter them out in the future work.

The last threat comes from the scale of our data set. As mentioned above, we only used six software projects systems to validate the effectiveness of MinClass. It is a very small scale when compared with the number of software projects in the whole software ecosystem. Thus, in the future, we need to replicate this work on a larger number of software projects.

V. CONCLUSIONS AND FUTURE WORK

This paper proposed an approach named MinClass to mine the key classes from software systems, with the aim to facilitate the process of developers to understand unknown software systems. Our approach mined the key classes by examining a very small number of classes. To this end, our approach treated any software system as a class coupling network, and proposed a new metric, *OSE*, to measure the importance of each class in the system. Empirical results on a set of six subject software projects showed that our MinClass performed best in the whole set of software systems, according to the average ranking of the Friedman test, when compared with eight other approaches in the baseline.

Our future work includes: (1) replicating this work on other non-Java software systems; (2) replicating this work on a larger number of software systems; and (3) proposing some new approaches to improve the performance of key class mining.

ACKNOWLEDGMENT

The authors would like to thank all the reviewers for their positive and valuable comments and suggestions regarding this manuscript.

REFERENCES

[1] L. E. G. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: An interview study with industry practitioners," *IEEE Trans. Softw. Eng.*, vol. 46, no. 4, pp. 346–361, Apr. 2020, doi: 10.1109/TSE.2018.2854716.

[2] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1125–1143, Dec. 2017, doi: 10.1109/TSE.2017.2655524.

[3] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using Web mining techniques," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 20, no. 6, pp. 387–417, Nov. 2008, doi: 10.1002/smr.370.

[4] W. Pan, H. Ming, C. Chang, Z. Yang, and D.-K. Kim, "ElementRank: Ranking java software classes and packages using a multilayer complex network-based approach," *IEEE Trans. Softw. Eng.*, early access, Oct. 8, 2019, doi: 10.1109/TSE.2019.2946357.

[5] F. Perin, L. Renggli, and J. Ressia, "Ranking software artifacts," in *Proc. 4th Workshop FAMIX Moose Reeng. (FAMOOSr)*, 2010, pp. 1–4.

[6] M. S. Wang, H. M. Lu, Y. M. Zhou, and B. W. Xu, "Identifying key classes using H-index and its variants," *J. Frontiers Comput. Sci. Technol.*, vol. 5, no. 10, pp. 891–903, 2011, doi: CNKI:SUN:KXTS.0.2011-10-005.

[7] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *Proc. 19th Work. Conf. Reverse Eng.*, Kingston, ON, Canada, Oct. 2012, pp. 93–102.

[8] J. Feng, D. Shi, and X. Luo, "An identification method for important nodes based on K-shell and structural hole," *J. Complex Netw.*, vol. 6, no. 3, pp. 342–352, Jul. 2018, doi: 10.1093/comnet/cnx035.

[9] W. Jiang and N. Dai, "Identifying key classes algorithm in directed weighted class interaction network based on the structure entropy weighted LeaderRank," *Math. Problems Eng.*, vol. 2020, Dec. 2020, Art. no. 9234042, doi: 10.1155/2020/9234042.

[10] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *J. Syst. Softw.*, vol. 110, pp. 28–53, Dec. 2015, doi: 10.1016/j.jss.2015.08.014.

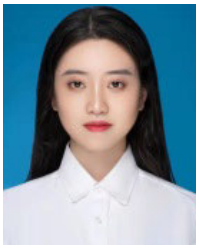
[11] J. Ai, W. Su, S. Zhang, and Y. Yang, "A software network model for software structure and faults distribution analysis," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 844–858, Sep. 2019, doi: 10.1109/TR.2019.2909786.

[12] M. H. Osman, M. R. V. Chaudron, and P. V. D. Putten, "An analysis of machine learning algorithms for condensing reverse engineered class diagrams," in *Proc. IEEE Int. Conf. Softw. Maint. (ICSM)*, Washington DC, USA, Sep. 2013, pp. 140–149.

[13] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proc. 22nd Int. Conf. Program Comprehension (ICPC)*, Hyderabad, India, 2014, pp. 110–121.

[14] P. Meyer, H. Siy, and S. Bhowmick, "Identifying important classes of large software systems through k-core decomposition," *Adv. Complex Syst.*, vol. 17, no. 8, Dec. 2014, Art. no. 1550004, doi: 10.1142/S0219525915500046.
[15] S. J. Jiang, X. L. Ju, X. Y. Wang, H. Y. Li, Y. M. Zhang, and Y. Q. Liu, "Measuring the importance of classes using UIO sequence," *Acta Electron. Sinica*, vol. 43, no. 10, pp. 2062–2068, Oct. 2015, doi: 10.3969/j.issn.0372-2112.2015.10.027.
[16] I. Ăžora and C.-B. Chirila, "Finding key classes in object-oriented software systems by techniques based on static analysis," *Inf. Softw. Technol.*, vol. 116, Dec. 2019, Art. no. 106176, doi: 10.1016/j.infsof.2019.106176.
[17] W. F. Pan, B. B. Song, K. S. Li, and K. J. Zhang, "Identifying key classes in object-oriented software using generalized K-core decomposition," *Futur. Gener. Comp. Syst.*, vol. 81, pp. 188–202, Apr. 2018, doi: 10.1016/j.future.2017.10.006.
[18] L. do Nascimento Vale and M. de Almeida Maia, "Key classes in object-oriented systems: Detection and assessment," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 10, pp. 1439–1463, Oct. 2019, doi: 10.1142/S0218194019500451.
[19] F. B. E. Abreu, G. Pereira, and P. Sousa, "A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems," in *Proc. 4th Eur. Conf. Softw. Maint. Reeng. (CSMR)*, Mar. 2000, pp. 13–22.
[20] F. Brito e Abreu and M. Goulao, "Coupling and cohesion as modularization drivers: Are we being over-persuaded?" in *Proc. 5th Eur. Conf. Softw. Maintenance Reeng.*, Lisbon, Portugal, Mar. 2001, pp. 47–57.
[21] W. Pan, B. Li, J. Liu, Y. Ma, and B. Hu, "Analyzing the structure of Java software systems by weighted K-core decomposition," *Future Gener. Comput. Syst.*, vol. 83, pp. 431–444, Jun. 2018, doi: 10.1016/j.future.2017.09.039.
[22] H. Li, X. Yang, Y. Li, L.-Y. Hao, and T.-L. Zhang, "Evolutionary extreme learning machine with sparse cost matrix for imbalanced learning," *ISA Trans.*, vol. 100, pp. 198–209, May 2020, doi: 10.1016/j.isatra.2019.11.020.

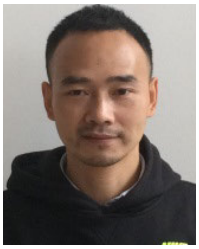
HAO LI is currently pursuing the M.S. degree with the School of Computer Science and Information Engineering, Zhejiang Gongshang University. His research interests include software engineering and complex networks.



TIAN WANG is currently pursuing the M.S. degree with the School of Computer Science and Information Engineering, Zhejiang Gongshang University. Her research interests include software engineering and complex networks.

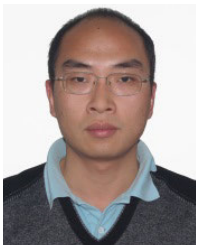
WEIFENG PAN received the Ph.D. degree from the School of Computer, Wuhan University, China, in 2011. He has been a Visiting Scholar with Western Michigan University. He is currently an Associate Professor and the M.S. Supervisor with the School of Computer Science and Information Engineering, Zhejiang Gongshang University. He has published more than 50 articles in international journals, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Future Generation Computer Systems*, and *Cluster Computing*. His current research interests include software engineering, service computing, complex networks, and intelligent computation. He is also a member of the China Computer Federation (CCF) and the CCF Service Computing Association.

MUCHOU WANG received the bachelor's degree from the Department of Computer Science, Zhejiang University of Technology (ZJUT), in 2005, and the master's degree from the Huazhong University of Science and Technology (HUST), in 2008. He is currently working with Wenzhou University. His research interests include service-oriented software engineering and digital library.



CHUNLAI CHAI received the M.S. degree from the School of Computer, Hohai University, China. He is currently an Associate Professor and the M.S. Supervisor with the School of Computer Science and Information Engineering, Zhejiang Gongshang University. He has published more than ten articles in international journals. His current research interests include software engineering and intelligent computation.

PENGYU CHEN is currently pursuing the bachelor's degree with Hangzhou Dianzi University, China. His current research interests include software engineering and data mining.



JIALE WANG received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is currently an Associate Professor with the School of Computer Science and Information Engineering, Zhejiang Gongshang University. He has published more than 20 articles in international journals and conferences. His current research interests include software engineering, service computing, and intelligent algorithms.

JING WANG received the Ph.D. degree from the School of Computer, Wuhan University, China. He is currently an Associate Professor with the School of Software and Internet of Things Engineering, Jiangxi University of Finance and Economics. He has published more than 25 articles in international journals and conferences. His current research interests include intelligent optimization algorithm and scheduling optimization. He is also a member of the China Computer Federation (CCF).