Journal of Engineering Science and Technology
Vol. 16, No. 4 (2021) 3358 – 3386
© School of Engineering, Taylor's University

JaCoCo-COVERAGE BASED STATISTICAL APPROACH FOR RANKING AND SELECTING KEY CLASSES IN OBJECT-ORIENTED SOFTWARE

BILAL I. AL-AHMAD[1,*], ISMAIL AL-TAHARWA [1], RAMI S. ALKHAWALDEH[1], IYAD M. ALAZZAM[2], NAZEEH GHATASHEH[3]

[1]Department of Computer Information Systems, The University of Jordan, Aqaba, Jordan
[2]Department of Information Systems, Yarmouk University, Irbid, Jordan
[3]Department of Information Technology, The University of Jordan, Aqaba, Jordan
*Corresponding Author: b.alahmad@ju.edu.jo

Abstract

Ranking and selecting the most fundamental classes in a software is a critical task for recognizing an inexperienced system. There are several approaches to detect these classes. This paper proposes a statistical-rank based approach that addresses an untouched area of testing coverage usefulness. Testing coverage information is used as new method to select the key classes in an object-oriented software. The proposed approach comprises three statistical phases including ranking the software classes, selecting the highly coverage classes, and selecting the key classes. Experiments are conducted on 23 successive releases of large open-source software system, Apache Lucene and three different projects are investigated to validate the proposed approach. A multiple linear regression model has been applied both to evaluate the proposed approach and to show the relationship between testing coverage of classes and their structural properties. The most significant predictors for testing coverage are complexity, coupling, lack of cohesion, size, and inheritance which constitute properties of the software classes. The testing coverage percentage of the key classes ranged between 72.7% - 82.6%. Our approach aims to help software engineers to assess software design and better achieve testing objectives.

Keywords: JaCoCo coverage weight, Key classes, Testing coverage metrics.

1. Introduction

Software testing is one of the processes in the software development life cycle. The main goal of the testing process is to ensure that the deliverable software meets the customer's specification and is developed according to certain standards and rules. Regression testing is done to ensure that the changes made on the software did not cause defects and to confirm that the previous features and functionalities are working properly. Regression testing is considered to be the main costly phase in the software testing process since typically it requires rerunning all test cases in all test suites. This takes a long time especially with a large number of test cases. For this reason, focusing on key classes in terms of testing coverage perspective would be helpful in mitigating testing cost [1, 2].

Improving the quality of software products has become an urgent need in the software development process. Software engineers find it hard to test the entire system especially with limited resources in existing complex software. Regression testing aims to improve the software quality and it has been the main costly phase in the software testing [3]. Therefore, software engineers strive to find an efficient technique to rank and select the most significant (or key) classes for understanding the whole software system and reducing the overall testing cost. In addition to improving the software quality and minimizing software testing efforts, key coverage-classes play an important role in improving Software Quality Assurance (SQA). New trends of SQA examine both the software product and process metrics in order to consolidate prediction obtained by conventional software engineering Key Performance Indicators (KPI). Code coverage enhances SQA efficiency by applying KPI metrics to the selected classes [4, 5]. Finding and pinpointing highly covered code segments of software engineering projects promote code re-usability trend of software engineering [6]. Furthermore, software testing results, in terms of code coverage, can act as a means to validate attained KPIs by software engineering projects [7].

The key classes are strongly coupled components of software system and capture the important functions that help software engineers early understand the key concepts during software maintenance [8]. Thus, focusing on the key classes with additional tests is necessary for improving the software quality. The importance of a class depends on multiple factors such as the complexity, coupling, and location among software system classes. The automatic finding of key classes was initially proposed by Zaidman and Demeyer [9]. The cost of understanding internal elements of software systems is related to the effort of investigating the software artifacts such as the source code [10]. Therefore, identifying key classes is an essential task in assessing software design and better attain software testing objectives in terms of cost. Several approaches attempt to find the key classes of software system; however, their approaches focus only on the aspects of the code metrics, their experiments are time consuming, and they do not pay an attention to the role of testing coverage in ranking and selecting key classes.

In this paper, we present a statistical approach which captures the testing coverage information to rank and select the key classes. The proposed approach uses the testing coverage as a baseline for ranking and selecting key classes in object-oriented software. To get the coverage information of classes, we use five code coverage metrics (instruction, branch, line, complexity, and method) obtained by the JaCoCo testing model. Code coverage is a common metric used as a benchmark to measure the efficiency of testing through implementing all the test cases that come along with

the software classes. The proposed approach quantifies the testing coverage at both class and release level. In the class-level, we calculate JaCoCo Coverage Weight (JCW) represented by the normalized average of the five code coverage levels. A higher value of JCW indicates a better coverage class. In the release-level, JCW is calculated to get the overall testing coverage weight for all classes within a release. Our approach uses testing coverage of classes and releases and includes three statistical phases: (1) Ranking all the classes of the software system by using the sequential ranking method [11], (2) Selecting the highly coverage classes by utilizing the box blot rule [12], and (3) Selecting the key classes by applying either empirical rule or Chebyshev's theorem [13] based on the normality of the classes coverage distribution . Experiments are performed on 23 successive releases of large open-source software system, namely, Apache Lucene and four extra releases of three different projects (Apache Ant, Apache POI, and net. bytebuddy). In this paper, there are four contributions that answer the following research questions:

- RQ1. What is the effectiveness of class testing coverage on ranking software classes?
- RQ2. What kind of relationship can be found between class testing coverage and its complexity?
- RQ3. How focusing on key classes would assess the software design quality?
- RQ4. How focusing on key classes would achieve better testing objectives?

The rest of this paper is organized as follows: Section 2 discusses the related work stated in the literature, Section 3 presents the proposed methodology, Section 4 shows results and discussions, and Section 5 presents conclusions and future work.

2. Related Work

Several approaches were used to identify the key classes from the perspective of source code comprehension. The approaches for selecting the key classes applied web mining [9, 14, 15], software network structure [16-20], machine learning [21-25], and code changing metrics [26, 27].

Some approaches applied dynamic analysis using web mining techniques to detect key classes of a software system, [9] and rank the important classes in a given software through using web mining combined with static coupling metrics. This approach identifies key classes by showing the classes that implemented the most controlling functionalities of software. Moreover, the approach [15] introduces a prototype called ROSE that uses web mining methods to detect the associated changes to help developers in making the required modifications. Also, [14] uses dynamic web mining analysis through applying the popular ranking algorithm (HITS). The disadvantages of these approaches are time consuming and counted different user scenarios.

Other approaches used software network structure features to discover key classes in software system. The approach [19] builds network from software elements and their relations, then the network is measured based on specified indicators and the classes are ranked according to results. The study [16] used many graph indexes to label key classes to detect key classes by applying eigenvector among centrality, closeness, and centrality to determine the significance location of given class in a software graph. Besides that, the approach [17] used the h-index instead of centrality indicators to detect the location of important classes. The results show that h-index has exceeded the performance in labelling the key

compared to the centrality indicators in the software graph. Also, K-core decomposition is applied on software graph by [18] to label important classes within the software, where the nodes located in the innermost layer are considered important compared with nodes located in the external layers. In addition, the approach [20] combines the page-rank algorithm with the design metrics through calculating dependency among design metrics and set different weights for direct edges in software graph. The main problem of these approaches is time consuming, and it is complicated to build such software network.

Different approaches use machine learning techniques for identifying the key classes. The study [21] introduces reverse engineered class diagrams to identify important classes of a system. Available forward design diagrams are used to learn and then to validate the quality of a set of prediction algorithms. The approach [22] uses learning algorithms with design metrics. In [23], the authors used the abstract syntax tree representations of the classes in order to determine the similar Java classes. Whereas the studies [24, 25] propose a new model to assess the testability of classes based on UML class diagram and they used two java programs for the evaluation process.

Some approaches applied different code changing criteria to identify the importance of classes in software system. In [28], the classes are identified according to the times that classes are changed together frequently. This approach focuses on the relations between the classes and the code implementation metrics. In [26], the authors used the change history of classes along with the class-level implementation metrics. Also, a technique introduced by [27], depends on structural dependencies between classes to detect key classes. This approach concentrates on static analysis of the source code. These approaches used only few numbers of dependencies metrics and did not take into account testing coverage metrics of the classes.

Current approaches do not pay attention to the role of code coverage metrics as a baseline for ranking and selecting key classes of software system, and they use few static metrics to assess the structural characteristics of the software classes. Our approach addresses these critical limitations through proposing a rank-based statistical approach that uses five testing coverage weights to measure the importance of the classes in terms of testability. Our approach employs testing coverage of both classes and releases of software system. It includes three main phases: ranking the software classes, selecting the highly coverage classes, and then selecting the key classes. Moreover, this approach attempts to assess the software design and improve software testing objectives.

3. Methodology

JaCoCo [29] was created by the EclEmma [30]. It is a free Java code coverage library that runs JUnit test cases and provides a visual coverage report by providing information for each single test case associated with each particular software module [31]. As in the study [32], JaCoCo is the most popular testing model that gives higher visibility and easier integration for testing Java modules. The coverage information is important during the regression testing to highlight the covered code portions while running the associated test cases [2, 33].

3.1. The proposed methodology

This paper introduces a statistical-ranked based approach that uses testing coverage information to rank and select key classes in object-oriented software. The

proposed approach comprises three statistical phases including (1) ranking the software classes, (2) selecting the highly coverage classes, and (3) selecting the key classes that exceed one standard deviation of the release coverage mean. Figure 1 illustrates the conducted steps of our proposed approach. Our approach has seven sequential steps described as follows:

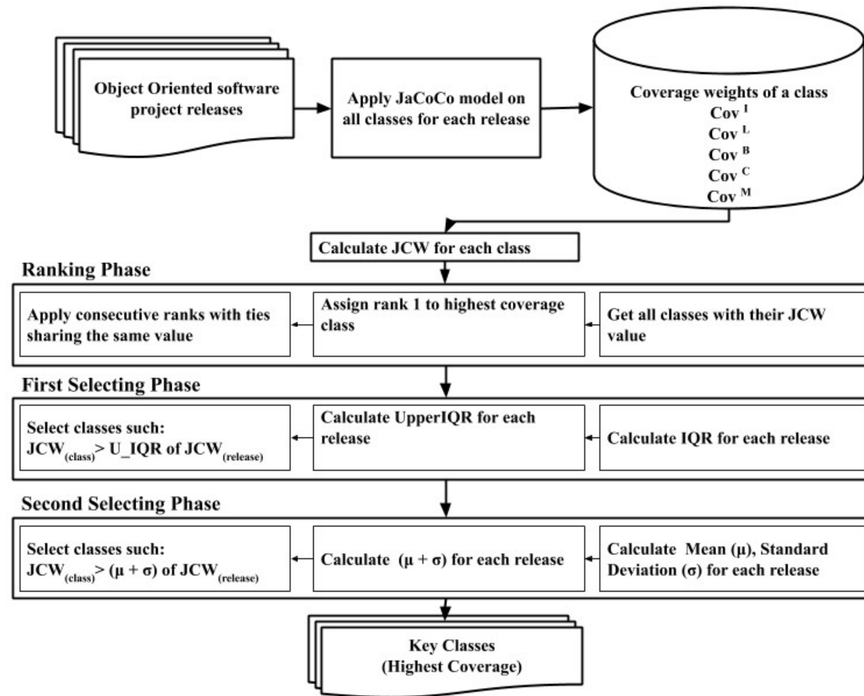

Fig. 1. The proposed approach.

- Collecting dataset, collecting object-oriented software releases which contain the classes and the associated test cases.
- Using JaCoCo model, applying JaCoCo testing model on all classes through all software releases to get five code coverage metrics (lines, branches, instructions, methods, and cyclomatic complexity).
- Getting testing coverage weight for each class, calculating testing coverage weights named as Cov^L, Cov^B, Cov^I, Cov^M, and Cov^C by using the proposed Equations: Eqs. (1)-(5) respectively. Then, get the JaCoCo Coverage Weight (JCW_{ci}) for each class by finding the normalized average of the five stated coverage weights.
- Ranking classes, ranking all classes in each release (JCW_{Rank}^{ri}) based on the values of (JCW_{ci}).The proposed approach ranks all classes by assigning rank 1 to the highest coverage class and uses sequential rank to unique values ties until the last ranked class.
- Getting testing coverage weight for each release, calculating testing coverage of each release (JCW_{ri}) which considers testing coverage of all classes within a release.

- Selecting the highly coverage classes, based on the calculated testing coverage at both class and release levels, the approach applies the boxplot rule to each software release. This step shows the first selection (JCW_{S1}^{ri}) of the highly coverage classes that exceeds the *Upper_IQR* ($Q3 + 1.5 * IQR$) value of the release coverage. We use boxplot rule since it is a standard statistical [34] method used to display the distribution of the software classes based on their testing coverage. A boxplot is a standardized way of displaying the distribution of data based on first quartile ($Q1$), third quartile ($Q3$), *IQR* (Inter Quartile Range), and *Upper_IQR* [32].
- Selecting the key classes, calculating the standard deviation and the mean of the release coverage for all resulting classes in the first selection phase. This final step identifies highly ranked classes (key classes) whose coverage surpasses one standard deviation of the release coverage mean; it represents the second selection phase (JCW_{S2}^{ri}) of the most important classes for each software release. We use standard deviation and mean of the release coverage to detect the distribution of the investigated classes through software releases. Our approach proposes five Equations: Eqs. (1)-(5), to quantify line coverage, branch coverage, instruction coverage, method coverage, and complexity coverage respectively. The Equations are defined as follows:

$$Cov_{(ci)}^{L} = \frac{CL_{(ci)}}{TNCL_{(ri)}} \quad (1)$$

where $Cov_{(ci)}^{L}$: Line coverage weight of the class, $CL_{(ci)}$: Covered Lines of class *ci*, and $TNCL_{(ri)}$: Total Number of all Classes' Lines in each release *ri*.

$$Cov_{(ci)}^{B} = \frac{CB_{(ci)}}{TNCB_{(ri)}} \quad (2)$$

where $Cov_{(ci)}^{B}$: Branch coverage weight of the class, $CB_{(ci)}$: Covered Branches of class ci, and $TNCB_{(ri)}$: Total Number of all Classes' Branches in each release *ri*.

$$Cov_{(ci)}^{I} = \frac{CI_{(ci)}}{TNCI_{(ri)}} \quad (3)$$

where $Cov_{(ci)}^{I}$: Instruction coverage weight of the class, $CI_{(ci)}$: Covered Instructions of class ci, and $TNCI_{(ri)}$: Total Number of all Classes' Instructions in each release *ri*.

$$Cov_{(ci)}^{M} = \frac{CM_{(ci)}}{TNCM_{(ri)}} \quad (4)$$

where $Cov_{(ci)}^{M}$: Method coverage weight of the class, $CM_{(ci)}$: Covered Methods of class *ci*, and $TNCM_{(ri)}$: Total Number of all Classes' Methods in each release *ri*.

$$Cov_{(ci)}^{C} = \frac{CC_{(ci)}}{TNCC_{(ri)}} \quad (5)$$

where $Cov_{(ci)}^{C}$: Complexity coverage weight of the class, $CC_{(ci)}$: Covered Complexity of class *ci*, and $TNCC_{(ri)}$: Total Number of all Classes' Complexity in each release *ri*.

The ranking, first selecting, and second selecting phases are conducted sequentially based on testing coverage at both class-level and release-level. The ranking phase uses testing coverage at class-level. The first selecting phase compares testing coverage of a class with the *Upper_IQR* value of its release

coverage, and the second selecting phase compares testing coverage of a class with one standard deviation of its release coverage mean. Algorithm 1 shows the pseudo-code of our proposed methodology. For example, assume that there exists a class *x* in release *y* (which has 10 classes), and associated with the test case *t*. After applying the test, class *x* has 100 covered lines, 5 covered branches,10 covered instructions, 6 covered methods, and 18 covered control flows in term of Cyclomatic Complexity. Suppose that the release y has 1200 lines, 50 branches, 100 instructions, 24 methods, 96 control flows, $U_IQR^{y}_{JCW}$= 5%, and $(\mu+\sigma)^{y}_{JCW}$= 13.8%. By using our proposed approach, we can quantify the weight for each coverage metric through detecting the coverage of class (*x*) in respect to its release (y) coverage as follows: $Cov^{L}_{(x)}$= 8.3%, $Cov^{B}_{(x)}$= 10%, $Cov^{I}_{(x)}$= 10%, $Cov^{M}_{(x)}$= 25%, and $Cov^{C}_{(x)}$ = 18.7%. The proposed approach ranks class *x* based on its corresponding value of JCW_x .

Then, the approach checks if class x is considered as a highly coverage class or not based on the value of *Upper_IQR* of the release *y* coverage. Consequently, class *x* was selected in the first selection phase (since 14.4% > 5%). Next, the approach ensures if the class x identified as a key class or not based on one standard deviation of the release y coverage mean. Similarly, class *x* was also chosen in the second selection phase (i.e., 14.4% > 13.8%). The following section describes the used metrics.

Algorithm 1 The pseudo-code of the proposed approach

$R \leftarrow r_i : \forall i \in \{1,2,\ldots,n\}$, Releases set
$C \leftarrow c_i : \forall i \in \{1,2,\ldots,m\}$, Classes set
JCW_{ci} :JaCoCo Coverage Weight of a single class c_i
JCW_{ri} :JaCoCo Coverage Weight of all classes in r_i
U_IQR_{ri} :Upper Inter Quartile Range of the release r_i
$JCW^{r_i}_{S1}$:First Selecting of highly coverage classes
$JCW^{r_i}_{S2}$:Second Selecting of the key classes
for $r_i \in R$ **do**
  for $c_i \in C$ **do**
    Use Equation 1 to calculate Cov^L for each class c_i
    Use Equation 2 to calculate Cov^B for each class c_i
    Use Equation 3 to calculate Cov^I for each class c_i
    Use Equation 4 to calculate Cov^M for each class c_i
    Use Equation 5 to calculate Cov^C for each class c_i
    $Cov_sum_{ci} = \sum_{i=1, j\in\{L,B,I,M,C\}}^{n} Cov_i^j$
    $JCW_{ci} = \frac{Cov_sum_{ci}}{5} \times 100\%$
  end for
  // Rank each class c_i based on JCW in each release
  $JCW^{r_i}_{Rank} = \{JCW_{c_i}; JCW^{c_i}_{Rank}\}$
  // Select highly coverage classes in each release
  $JCW^{r_i}_{S1} = \{JCW^{r_i}_{Rank}; JCW_{c_i} > U_IQR^{r_i}_{JCW}\}$
  // Select key classes in each release
  $JCW^{r_i}_{S2} = \{JCW^{r_i}_{S1}; JCW_{c_i} > (\mu+\sigma)^{r_i}_{JCW}\}$
end for

3.2. Metrics and tools

Software metrics are quantifiable measures of software characteristics. This study employs five testing coverage metrics to build the proposed approach, and it uses 21 static metrics to find the association between the testing coverage of classes and their structural properties. The definitions of these metrics are explained in the subsections 3.2.1 and 3.2.2.

3.2.1. Testing coverage metrics

The proposed approach uses JaCoCo code coverage tool which generates five testing coverage levels for classes after conducting the corresponding JUnit test cases [35]. The testing coverage metrics described as follows:

- Lines coverage: For all class files compiled with debug information, coverage information for individual lines can be calculated. A source line is covered when at least one instruction is assigned to an executed line. It reflects the amount of code that has been exercised based on the number of Java byte code instructions called by the JUnit test [36].
- Branches coverage: Counts branch coverage for all if and switch statements. This metric counts the total number of executed branches in a method within each class through implementing JUnit test [37].
- Instruction's coverage: Counts instructions into a single Java byte code. Instruction coverage represents the amount of code that has been executed by implementing JUnit test [38].
- Methods coverage: Each concrete method contains at least one instruction. A method is considered as executed when at least one instruction has been executed by implementing JUnit test [36].
- Cyclomatic Complexity coverage: Measures all possible control flows within each concrete method within each class. All the above metrics are derived from Java byte code instructions and debug information which existed in classes' files [39].

3.2.2. Static product metrics

In the context of our study, we use the most common static metrics in the literature [40-44] to obtain the relationship between the testing coverage of classes and their structural properties (21 static product metrics) [45-46]. These metrics are only used for model evaluation and are collected using Understand [47] and ckjm [48] tools.

The static metrics measures software quality attributes and described as follows:

- C and K metrics suite [49]
  - Weighted method per class (WMC): Represents sum of the cyclomatic complexities of all declared local methods within a class.
  - Depth of inheritance tree (DIT): Measures class level in the inheritance tree, root class is considered as zero.
  - Number of children (NOC): Represents to the number of immediate children's or sub classes of a given class in the hierarchy.

  - Coupling between object classes (CBO): Measure the total number of new or redefined methods to which all the inherited methods are coupled.
  - Response for a Class (RFC): Measures the total number of local methods plus the number of non-local methods called by local methods.
  - Lack of cohesion in methods (LCOM): Measures the dissimilarity of methods in a particular class that shared at minimum one particular field or attribute. Lack of cohesion increases the likelihood of getting more defects.

- Henderson-Sellers's metric [50]
  - Lack of cohesion in methods (LCOM3): It is an extension from LCOM metric; it represents the connected components in the graph.

- QMOOD metrics suite [51]
  - Number of Public Methods (NPM): Measures total number of all the public methods in a given class.
  - Data Access Metric (DAM): Represents proportion of the number of private or protected attributes to the overall number of declared attributes within a class.
  - Measure of Aggregation (MOA): Measures the total number of aggregation relationship between the class attributes.
  - Measure of Functional Abstraction (MFA): Represents percentage of number of inherited methods of an investigated class to the total number of methods that accessed by member methods of the class.
  - Cohesion Among Methods of Class (CAM): Computes the relatedness among methods of a class based on the parameters or specifications list of the methods.

- Tang metrics [52]
  - Inheritance Coupling (IC): Indicates the number of parent or super classes coupled with a particular investigated class.
  - Coupling Between Methods (CBM): It represents the number of classes with which the investigated class is coupled.
  - Average Method Complexity (AMC). This metric measures the average method size for each class such as Java byte code length in the method.

- Martin coupling metrics [53]
  - Afferent couplings (CA): Counts the number of classes calling the investigated class.
  - Efferent couplings (CE): Counts the number of classes called by the investigated class.

- McCabe's complexity metrics [54]
  - McCabe's cyclomatic complexity (CC): Total number of all potential various paths for a given method within a class. It represents method's control flow.
  - MAX (CC): The maximum value of CC among all methods within a particular class.
  - AVG (CC): The arithmetic means of all CC methods within a particular class.

- Lines of Code [55]
    - o Lines of code (LOC): Refers to the number of code lines for each class except the comments. Generally, a class with higher LOC number tends to be more complex than other classes.

4. Results and Discussions

This study involves two experiments. The first experiment is conducted on 23 successive releases of large open-source software system, Apache Lucene. The second experiment investigates three different size projects to validate the proposed approach. This section describes the selected dataset and the experimental setup which involves displaying the results in all phases (ranking all classes, selecting highly coverage classes, and selecting the key classes). Also, it shows model evaluation, software testing objectives, model validation, and threats to validity subsections respectively.

4.1. Dataset description

To validate the ability of our approach in selecting key classes in object-oriented software, we conducted experiments on large dataset of Apache Lucene project. The dataset contains 23 successive releases of Apache Lucene project, and three different projects (Apache Ant, Apache POI, and Maven net. bytebuddy). Table 1 presents statistical description of our dataset in each release as follows: (1) total number of classes, (2) total number of lines of code, and (3) testing coverage statistics in terms of mean, standard deviation, and *Upper_IQR*. With respect to size, $R23$ is the largest release with a total of roughly 999347 lines of code, while $R1$ is the lowest one with about 886307 lines of code. Also, the total number of classes varies between 3195 and 3402 through all investigated releases. Considering the mean, standard deviation, *Upper_IQR* of the releases, the coverage values remain within tiny ranges such (11.5%-12.3%), (8.7%-9.4%), and (5.1%-5.6%) respectively. In this study, we adopted the five testing coverage metrics denoting: Cov^l, Cov^i, Cov^m, Cov^b, and Cov^c to assess five coverage levels of a class.

Figure 2 shows the overall coverage ratio of each metric for each software release. Based on the results, the coverage values listed from highest to lowest values: (1) lines coverage (87.9%); (2) Instruction coverage (86.3%); (3) methods coverage (86.1%); (4) branches coverage (72.4%); and (5) complexity coverage (66%).

It is worth noting that for each coverage metric, the coverage generally remains stable over most releases except for $R22$. Take Cov^L as an example, its values keep within a small range between 86.8% and 88.2%. Likewise, the Cov^C values also remain in a minor range between 64.7% and 66.3%, and the Cov^B values fall within 71% and 72.7%.

But, with respect to instruction and method coverage (CovI and CovM), both have roughly the same coverage values. This is due to the fact that each method contains at least one executed instruction. Besides, the CovI values stay in a range between 85.2% and 86.6%. Similarly, the CovM values are kept within a slight range between 85% and 86.4%. Compared to other software releases, it is worth mentioning that the release (R22) has a slightly significant decline in all testing coverage aspects. This decrease indicates that it may include new features, architecture changes, or product components changes. Obviously, these minor

changes would most likely affect the testing coverage. The major release (R22) has 3341 classes compared with R21 which has 3372 classes.

Table 1. Dataset description.

No.	Release	No. of classes	LOC	Mean	Standard Deviation	Upper_IQR
R1	5.2.0	3198	886307	0.12306	0.09378	0.05593
R2	5.2.1	3195	922481	0.12263	0.09435	0.05607
R3	5.3.0	3299	952871	0.11934	0.09098	0.05464
R4	5.3.1	3300	953087	0.11908	0.09188	0.05470
R5	5.3.2	3304	953087	0.11925	0.09113	0.05441
R6	5.4.0	3369	973057	0.11663	0.09016	0.05261
R7	5.4.1	3350	973057	0.11698	0.09064	0.05326
R8	5.5.0	3345	973157	0.11699	0.09074	0.05341
R9	5.5.1	3373	978192	0.11629	0.09003	0.05264
R10	5.5.2	3375	978409	0.11602	0.09017	0.05257
R11	5.5.3	3369	978417	0.11605	0.09039	0.05262
R12	5.5.4	3369	978694	0.11663	0.09016	0.05261
R13	6.0.0	3375	929278	0.11588	0.09048	0.05256
R14	6.0.1	3215	929577	0.12377	0.09343	0.05548
R15	6.1.0	3217	929577	0.12389	0.09245	0.05535
R16	6.2.0	3271	961366	0.12166	0.09118	0.05539
R17	6.2.1	3299	961829	0.12191	0.09202	0.05439
R18	6.3.0	3310	970974	0.12068	0.09076	0.05390
R19	6.4.0	3365	986721	0.11932	0.08943	0.05340
R20	6.4.1	3366	986608	0.11916	0.08962	0.05303
R21	6.4.2	3372	986673	0.11965	0.09014	0.05305
R22	6.5.0	3341	999311	0.11583	0.08722	0.05218
R23	6.5.1	3402	999347	0.11622	0.08721	0.05194

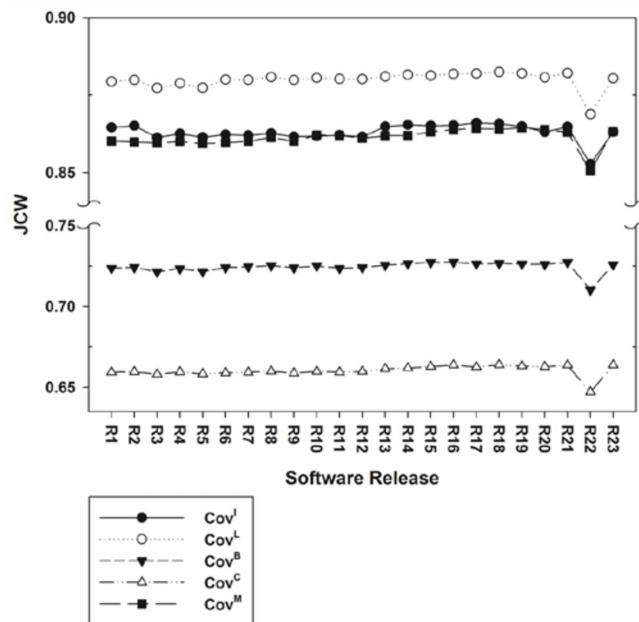

Fig. 2. The coverage metrics ratios per release.

4.2. Experimental setup

The experiments were conducted on 23 successive releases of Apache Lucene project and have three essential phases: (1) ranking the whole classes in each release based on their testing coverage; (2) selecting the highly coverage classes greater than the *Upper_IQR* of the release coverage; (3) from the resulting classes in second phase, selecting key classes in which their coverage exceeds one standard deviation of the release coverage mean. Table 2, 3, and 4 describe the results after conducting each phase sequentially. This subsection is related to the three essential phases of our approach which presents the findings associated with each phase.

4.2.1. Ranking the software classes

In the context of our approach, the first phase includes ranking all the classes in all software releases based on the associated JCW. Table 2 shows all the ranking classes in each release with their corresponding coverage, the *IndexWriter* ranked as the first class in each release, it has the highest testing coverage among all classes, and its values stay in a significant range between 72.7% and 82.6%. It should be emphasized that this complex class is widely tested than other classes. As well, this key class is a fundamental class used by many classes from the two essential Lucene packages, namely, *org.apache.lucene.index* and *org.apache.lucene.search*. The classes in the first package used it to maintain and access indexes while the classes in the second package used it to search through indexes. The Apache Lucene project which is critical for both searching and indexing of the documents over the web. On the other hand, the lowest ranked class differs from one release to another. Among all studied releases, all the lowest ranked classes have basically stable in extremely low coverage value (i.e., less than 1%) as shown in Table 2. For example, *BinaryEntry* class ranked as the last class for the first two releases with highly low coverage (0.13 %).

Similarly, *IntersectVisitor* ranked as the last class in the last eight releases with range 0.11% to 0.12%. These non-key classes are insignificant and represent unessential functionality of software system and are most likely allocated to a brief testing. It should be pointed out that the highly ranked classes are fundamental components that are generally invoked or inherited from many other classes. By contrast, the lowly ranked classes are specific and independent ones. The experiments show the value of code coverage metrics in ranking classes. The highly ranked class is considered as a key class since it has a highly testing coverage weight and in would reflect higher quality class. The results answer RQ1 to clarify how testing coverage used as an effective way for ranking of software classes.

Table 2. Ranking classes in all software releases.

Release	No. of classes	Highest ranked class	JCW	Class rank	Lowest ranked class	JCW	Class rank
5.2.0	3198	IndexWriter	0.77168	1	BinaryEntry	0.00131	2525
5.2.1	3195	IndexWriter	0.79786	1	BinaryEntry	0.00131	2523
5.3.0	3299	IndexWriter	0.75451	1	NormsEntry	0.00127	2607
5.3.1	3300	IndexWriter	0.79911	1	SortedSetEntry	0.00127	2609
5.3.2	3304	IndexWriter	0.76788	1	NormsEntry	0.00127	2613
5.4.0	3369	IndexWriter	0.75468	1	Norms	0.00124	2678
5.4.1	3350	IndexWriter	0.76465	1	Predicate	0.00125	2659
5.5.0	3345	IndexWriter	0.76465	1	Lock	0.00125	2659
5.5.1	3373	IndexWriter	0.75926	1	LockFactory	0.00124	2676
5.5.2	3375	IndexWriter	0.76748	1	RateLimiter	0.00124	2674

5.5.3	3369	IndexWriter	0.77301	1	PathNode	0.00124	2680
5.5.4	3369	IndexWriter	0.75468	1	FileEntry	0.00124	2678
6.0.0	3375	IndexWriter	0.78118	1	OneGroup	0.00124	2682
6.0.1	3215	IndexWriter	0.82648	1	Accountable	0.00128	2560
6.1.0	3217	IndexWriter	0.79646	1	QueryTimeout	0.00128	2563
6.2.0	3271	IndexWriter	0.79593	1	IntersectVisitor	0.00123	2606
6.2.1	3299	IndexWriter	0.81213	1	IntersectVisitor	0.00121	2618
6.3.0	3310	IndexWriter	0.79902	1	IntersectVisitor	0.00121	2637
6.4.0	3365	IndexWriter	0.77446	1	IntersectVisitor	0.00118	2673
6.4.1	3366	IndexWriter	0.77187	1	IntersectVisitor	0.00118	2674
6.4.2	3372	IndexWriter	0.79612	1	IntersectVisitor	0.00118	2679
6.5.0	3341	IndexWriter	0.72767	1	IntersectVisitor	0.00118	2655
6.5.1	3402	IndexWriter	0.75781	1	IntersectVisitor	0.00116	2708

4.2.2. Highly coverage classes

For the second phase, Figure 3 shows the distribution of classes based on first quartile (*Q*1), third quartile (*Q*3), *IQR*, and *Upper_IQR* of the testing coverage. In general, all releases have approximately very similar coverage values as 0.40%, 2.3%, and 1.8% for of *Q*1*, Q*3, and *IQR* respectively. According to *Upper_IQR* values, there is some little fluctuations from early releases to the later ones. There exists a slightly significant decrease between *R*2 and *R*3, however from *R*3 until *R*13, they have a very closed similar coverage. Also, there is a little significant increase from *R*14 to *R*18. Among all releases, *R*16 has the highest coverage release and *R*10 has the lowest coverage one. Obviously, there is no normal distribution of the classes, the distribution shows skewness to the left. In other words, most of the classes spread out under the coverage mean.

For the first release (*R*1), the total number of classes is 3198, 800 classes distributed under the first quartile within the highly minor coverage range between 0.13% and 0.46%. There are 1598 classes distributed between first and third quartiles within a range from 0.46% to 2.3%. In addition, there are 495 classes distributed between third quartile and *Upper_IQR* close to this range (2.3%-5.2%). But, only 305 classes are distributed greater than *Upper_IQR* of a release and are considered as a more highly testing coverage than the rest of the classes. In other words, only 9.5% of the total number of classes are considered as highly coverage classes in this particular release.

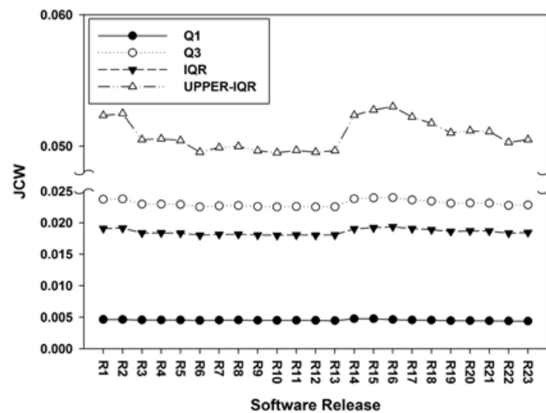

Fig. 3. Distribution of classes based on *Q*1, *Q*3, *IQR*, and *Upper_IQR*.

For the release *R*10, the whole investigated classes are 3375, 844 classes disseminated under *Q*1 within an extremely minor range (0.12%-0.44%). Similarly, there are 1687 classes circulated between *Q*1 and *Q*3 in a highly insignificant range such (0.44%-2.2%). In addition, there are 531 classes fallen between *Q*3 and *Upper_IQR* within a slightly significant range (2.2%-4.9%). But only 313 classes are distributed larger than *Upper_IQR* of the *R*10 coverage (4.9%). To be specific, this release has about 9.2% as highly coverage classes compared with the total number classes of software release. For the release *R*16, the whole investigated classes are 3271, 817 classes distributed under *Q*1 inside a highly insignificant coverage range from 0.12% to 0.46%. Also, there are 1636 classes allocated between *Q*1 and *Q*3 within coverage range (0.46%-2.3%). In addition, there are 526 classes scattered between *Q*3 and *Upper_IQR* within a marginally significant coverage range from 2.3%- 5.3%. Nonetheless, only 292 classes disseminated greater than *Upper_IQR* of release coverage (5.3%). Table 3 shows the selected classes with their corresponding coverage. Consequently, this release contains 8.9% outliers' classes with respect to the overall number of classes. Since *IndexWriter* ranked as the first class in each release in all phases, it has the highest testing coverage which constantly remains with a range between 72.7% and 82.6% as described above. Table 3 shows only the total number of selected classes after conducting the second phase with the lowly ranked classes for each release. For example, *PriorityQueue* ranked as the last in *R*3, *R*4, *R*5, and *R*6 with a minor coverage range from 5.0% to 5.4%. Likewise, *VisitorTemplate* ranked as the last class in seven releases *R*2, *R*6, *R*9, *R*10, *R*12, *R*13, *R*15 and its values fall within 4.9% and 5.6%.

Table 3. Selected classes after *UPPER_IQR*.

Release	No. of classes	Lowest ranked class	JCW	Class rank
5.2.0	305	SegmentMerger	0.05239	299
5.2.1	306	VisitorTemplate	0.05618	300
5.3.0	309	PriorityQueue	0.05100	303
5.3.1	312	PriorityQueue	0.05489	306
5.3.2	309	PriorityQueue	0.05096	303
5.4.0	312	VisitorTemplate	0.05311	306
5.4.1	314	PriorityQueue	0.05348	308
5.5.0	314	ContainsVisitor	0.05030	308
5.5.1	313	VisitorTemplate	0.04965	307
5.5.2	313	VisitorTemplate	0.04960	307
5.5.3	313	ContainsVisitor	0.05262	307
5.5.4	312	VisitorTemplate	0.05311	306
6.0.0	313	VisitorTemplate	0.05026	307
6.0.1	296	Utility	0.05293	299
6.1.0	297	VisitorTemplate	0.05279	291
6.2.0	292	TermAutomatonQuery	0.05308	286
6.2.1	296	DocumentsWriterDeleteQueue	0.05222	290
6.3.0	300	ConjunctionDISI	0.05176	294
6.4.0	305	ConjunctionDISI	0.05101	299
6.4.1	299	NorwegianLightStemmer	0.05122	293
6.4.2	300	RadixSelector	0.05117	294
6.5.0	306	Config	0.05060	300
6.5.1	304	ByteArrayDataInput	0.05071	298

Additionally, all the classes selected in Table 3 have coverage range within 4.9% and 5.6%. For example, *ConjunctionDISI* and *NorwegianLightStemmer* classes have 5.1% coverage and ranked as the last classes in the releases *R*18, *R*19, *R*20 respectively.

4.2.3. Key classes

For the third phase, Table 4 shows the selected key classes after conducting this phase. The proposed approach captures only the selected classes which resulted in the second phase. As a result, the number of the investigated class is less than in the second phase. Figure 4 shows the distribution of dataset based on mean (μ) and standard deviation (σ) of the software release coverage. The investigated releases have slightly significant ranges (11.5%-12.3%), (8.7%-9.4%), (20.3%-21.7%) for mean, standard deviation, and one standard deviation of the mean, respectively.

Since the approach focuses on getting all classes that exceed one standard deviation of the mean. The results clearly show that $R14$ has the highest coverage (21.7%) and $R22$ has the lowest coverage (20.3%). As With regard to $R14$, there are 296 selected classes, 205 classes scattered between ($\mu - \sigma$)and the mean within slightly significant coverage range from 2.7% to 12.2%. Additionally, 57 classes disseminated between mean and one standard deviation of the mean, within significant coverage range between 12.2% and 21.7%.

Also, only 34 classes were distributed directly greater than one standard deviation of the release coverage mean. It is important to point out that the key classes only form 1% out of the whole number of the classes.

Table 4. Selected classes after one standard deviation of the mean.

Release	No. of classes	Lowest ranked class	JCW	Class rank
5.2.0	34	BrazilianStemmer	0.21957	28
5.2.1	33	CompressingTermVectorsWriter	0.21713	27
5.3.0	34	CompressingTermVectorsWriter	0.21737	28
5.3.1	34	CompressingTermVectorsWriter	0.21699	28
5.3.2	34	CompressingTermVectorsWriter	0.21824	28
5.4.0	32	BrazilianStemmer	0.20907	26
5.4.1	33	CompressingTermVectorsWriter	0.21502	27
5.5.0	33	BrazilianStemmer	0.21312	27
5.5.1	32	StandardSyntaxParserTokenManager	0.21502	26
5.5.2	33	BrazilianStemmer	0.21403	27
5.5.3	31	StandardSyntaxParserTokenManager	0.21143	25
5.5.4	31	StandardSyntaxParserTokenManager	0.21929	25
6.0.0	33	BrazilianStemmer	0.20975	27
6.0.1	34	BrazilianStemmer	0.22414	28
6.1.0	34	BrazilianStemmer	0.22175	28
6.2.0	35	CompressingTermVectorsWriter	0.21909	29
6.2.1	36	JapaneseTokenizer	0.21738	30
6.3.0	37	BrazilianStemmer	0.21450	31
6.4.0	37	CompressingTermVectorsWriter	0.21071	32
6.4.1	39	JapaneseTokenizer	0.21230	30
6.4.2	36	JapaneseTokenizer	0.21330	30
6.5.0	39	StandardSyntaxParserTokenManager	0.20929	33
6.5.1	37	JapaneseTokenizer	0.20707	31

With reference to $R22$, there are 306 selected classes, 215 classes of all classes fall between ($\mu - \sigma$) and the mean with (2.7%-11.6%) coverage range, 52 classes distributed between the mean and one standard deviation of the mean with a coverage range (11.6%-20.5%), and only 39 classes fall above than one standard deviation of the mean. Particularly, 1.2% of the total number of classes is considered as key classes in this release. Table 4 shows the total number of selected classes after calculating one standard deviation of the mean for each release and displays the lowest ranked class with their corresponding coverage. For example,

CompressingTermVectorsWriter ranked as the last class in the releases *R*2, *R*3, *R*4, *R*5, *R*7, *R*16, and *R*19 within significant coverage range between 21% and 21.9%. By the same token, *JapaneseTokenizer* ranked as the lowest ranked class in the releases *R*17, *R*20, *R*21, and *R*23, its values keep within a small coverage range from 20.7% to 21.7%.

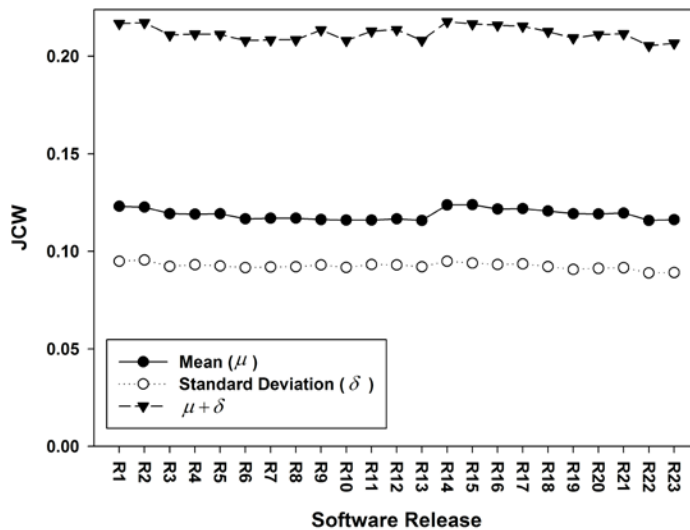

Fig. 4. Distribution of classes based on mean and standard deviation.

The proposed approach efficiently uses the testing coverage of a class as a baseline for ranking and selecting software system classes. It considers the classes selected in the above table as key classes for each associated release. As an example, *R*22 has 39 key classes, *R*11 and *R*12 has 31 key classes. Since this open-source project has sequential releases with a large number of common classes, it is necessary to find out the mutual key classes in all releases. It is worth mentioning that key classes demonstrate the key essential functionalities of software.

Additionally, one of the most important benefits of our approach is to find out the similar key and non-key classes in software system. Compared with the approach [23], our approach discovers seven key classes which remain stable overall releases, namely, KStemData1, KStemData2, KStemData3, KStemData4, KStemData5, KStemData6, and KStemData7. The coverage ranges for these classes fall within 26.8% and 29.1% across all releases, these key classes possibly called and inherited from many other classes. On the contrary, the total number of non-key classes differs from one release to another. For example, the first release has 871 similar classes within a coverage range from 0.13% to 5.1%, the *BinaryEntry* class is at the lowest (2525th) rank with 0.13%, and *CompressedBinaryTermsEnum* class is at (311th) rank with 5.1%. Also, the ratio of similarity of non-key classes in *R*1 is approximately 27.2% out of the total number of classes. These non-key classes are possibly not used by the other classes.

Lucene can index practically any type of text-containing document, it has become popular for use in Internet search engines. Lucene comprises four core

functionalities: searching, indexing, stemming, and tokenization. Lucene [56] affords strong searching and indexing through the documents over the web.

It applies strong efficient searching algorithms written in Java, the primary goal of Lucene [57] is to simplify information retrieval. Lucene enables searching by keywords. Lucene converts searching query text into certain fundamental indexed terms. In due time, these terms are used to identify which documents match the searching query.

Figure 5 illustrates top 30 mutual key classes among all releases. According to the obtained results, most of the key classes belong to indexing, stemming, and query parsing functionalities, which reflect the critical importance of these functionalities in facilitating searching over the web. As a result, our approach highlights these classes as the highest testing coverage and ranks them at the top of the resting classes. The highly ranked classes are fundamental classes that are generally invoked or inherited from many other classes. The overall results of classes rank for Apache Lucene confirm that core classes are ranked high, and non-core classes are ranked low.

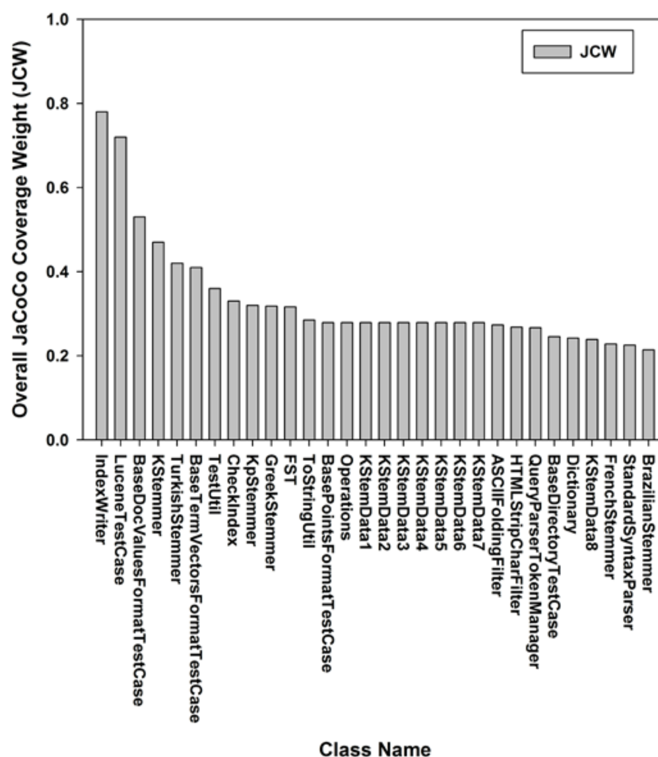

Fig. 5. Top 30 common key classes.

Among all common key classes, *IndexWriter* has the highest coverage class (78%). This class implements the most essential concept of Lucene since it creates, and updates indexes required for searching. It is frequently called by the classes of two key Lucene packages: *org.apache.lucene.index* and *org.apache.lucene.search*. Also, *BrazilianStemmer* has the lowest coverage (21.4%). *LuceneTestCase* class

ranked as the second key class, and it is a super class for all Lucene test case classes such the following key subclasses: *BaseDocValuesFormatTestCase*, *BaseTermVectorsFormatTestCase*, *BasePointsFormatTestCase*, and *BaseDirectoryTestCase*. Remarkably, our results show that key classes are complex and larger classes which demonstrate the most important functionalities of software.

This implies that Junit testing mainly focuses on the key classes since they are tightly coupled with the large number of non-key classes, frequently inherited by other classes, and low cohesive. Our approach highlights that focusing on most important classes could satisfy better testing objectives and assess the software design. The key classes which correlated to the key functionalities grouped as: (1) indexing (*IndexWriter, LuceneTestCase, BaseDocValuesFormatTestCase, BaseTermVectorsFormatTestCase, CheckIndex, BasePointsFormatTestCase*), (2) stemming (*KStemmer, KpStemmer, TurkishStemmer, GreekStemmer, FrenchStemmer, BrazilianStemmer, KStemData*1*, KStemData*2*, KStemData*3*, KStemData*4*, KStmData*5*, KStemData*6*, KStemData*7*, KStemData*8*, Dictionary*), and (3) queryparsing (*QueryParserTokenManager,StandardSyntaxParser*). All the eight KStemdata classes (*KStemData*1*, KStemData*2*, KStemData*3*, KStemData*4*, KStemData*5*, KStemData*6*, KStemData*7*, and KStemData*8) used by the *KStemmer* class. This class implements the Kstem algorithm for English language tokens. Among all stemmer's classes, *KStemmer* ranked as the highest coverage class due to the reason that English is the most common searching language over the web, and this reflects its importance.

It is important to note that changing test cases would lead to change the coverage characteristics of the classes and this might affect the results. Updating test cases it might affects the test coverage either positively or negatively, it positively affects when the created new test cases are done on the criterion that subsumes previous test cases criterion [58-60]. For example, if the new test cases are created to achieve prime path test criterion which are not covered by the previous test cases, this test case will increase the coverage. On the other hand, if the new test cases are created based on criterion less than previous test cases criterion, the impact of these test cases will be negative. For example, if we have already test cases created on prime path criterion and the new test cases are created based on simple path criterion and since the simple path is sub path of prime the coverage will be mitigated.

4.3. Model evaluation

To evaluate the proposed approach, multiple linear regression [61] is applied to show the relationship between the testing coverage of classes and their structural properties. For each release, multiple linear regression presents the correlation between the 21 independent static metrics (independent variables) and testing coverage (dependent variable). Figure 6 shows coefficient of determination and standard error of estimate through all investigated releases. In general, there is an increase of correlation from the earlier release to later releases. In light of the evaluation, $R7$ has the highest correlation with 76%, and lowest estimate error with 2.2%. This reflects that testing coverage of a particular class can be predicted with most certain static metrics. Also, there is a significant increase of correlation from $R5$ to $R6$. Whereas all the releases from $R6$ until $R23$ generally have steady significant correlation whose values remain

within the range between 68% and 74%. In addition, the releases have a constant low estimate error which is about 2.4% on average.

On the other hand, *R*3 and *R*5 have the lowest value of correlation (41%), and highest estimate error (3.8%). Obviously, there is significant oscillation in the earlier releases from *R*2 to *R*5. This variation occurred due to features updating that occurred with the release's pairs (*R*2, *R*3) and (*R*5, *R*6) would probably affect the structural properties of the classes in these particular releases. Clearly, *R*2 has 3195 classes and *R*3 has 3299 classes, the difference in the number of classes points out to the features changing between *R*2 and *R*3, and in the same way between *R*5 and *R*6. There are a set of significant static metrics capable of precisely defining which classes should be considered as key ones or not, because there is a significant correlation between testing coverage and static product metrics. This indicates that the increase in testing coverage is positively correlated with complexity, coupling, lack of cohesion, size, and inheritance properties of classes.

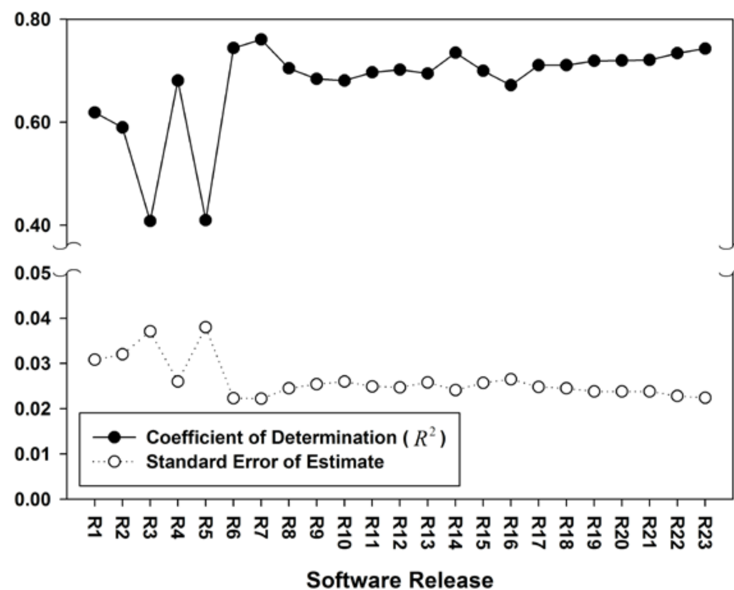

Fig. 6. Evaluation of multiple linear regression.

In fact, the results point out that a large class containing a large number of methods and attributes, a large number of control flow paths for methods. Consequently, it is an important class compared with non-key classes. This may be due to the fact that larger complex class is also more likely to be more extensively tested. The results identify the positive relationship that can be found between class testing coverage and its complexity. A higher testing coverage class expresses a higher complexity in terms of large number of (longer methods, control paths, and lines of code). The experiments show how the higher of testing coverage of class indicates a higher degree of class complexity. Key classes are often more complex and larger than non-key classes.

Table 5 shows that the most significant predictors of testing coverage such as: (1) complexity(CC, WMC, AMC); (2) coupling (CBO, CE, RFC); (3) Lack of

cohesion (LCOM); (4) size (LOC); and (5) inheritance (DIT). Those metrics are the most frequently common predictors across all the investigated software releases. Since the results shows that the complexity metrics of class (CC, WMC, AMC) are strongly correlated with its testing coverage, this finding responds to the issue of RQ2. To obtain the most significant correlated metrics among all studied metrics, comparing of means test is applied to show difference between key classes and non-key classes groups. As an example, for *R*1, the results confirm significant difference between key and non-key classes as shown in Table 6. Also, findings have revealed that the key classes are tightly coupled with other classes and tend to lack cohesion. The results show the significant correlation between testing coverage of classes and its design properties. Remarkably, key coverage class mirrors a higher coupling and low cohesion values.

In general, these key classes are tightly coupled with other classes and have dissimilar methods within a class. This improves the understanding of the design architecture and helps to early detect the critical design problems. Key classes have high coupling and low cohesion comparing with non-key classes. Moreover, key coverage classes give the impression to have higher priority due to its higher impact on the overall design. Thus, the complexity associated with these key classes can be related to the fact that classes are more involved in the software design. Also, mean values for coupling and cohesion metrics for key coverage classes are significantly worse than for non-key coverage classes.

Table 5. Significant predictors of testing coverage per each release.

Release	LOC	CC	AMC	CBO	LCOM	WMC	CE	DIT	IC	RFC	MFA	NPM	MOA	NOC
5.2.0	X	X	X	X	X	X	X	X		X				
5.2.1	X	X	X		X	X		X					X	X
5.3.0	X	X	X								X			X
5.3.1	X	X	X	X	X	X	X	X		X				
5.3.2	X	X	X		X		X		X	X	X	X	X	X
5.4.0	X	X	X	X	X	X	X	X				X		X
5.4.1	X	X	X	X	X	X	X	X		X	X	X		
5.5.0	X	X	X	X	X	X	X	X			X	X		X
5.5.1	X	X	X	X	X	X	X	X				X		
5.5.2	X	X	X	X	X	X		X		X				X
5.5.3	X	X	X	X		X		X		X		X		
5.5.4	X	X	X	X		X	X	X		X		X	X	X
6.0.0	X	X	X	X	X	X		X	X	X			X	
6.0.1	X	X	X	X	X	X		X	X	X			X	
6.1.0	X	X	X	X	X	X	X	X	X	X				
6.2.0	X	X	X	X	X			X		X			X	X
6.2.1	X	X	X	X	X	X	X	X	X	X		X		X
6.3.0	X	X	X	X	X	X	X		X	X		X	X	X
6.4.0	X	X	X	X	X	X	X		X	X		X	X	

It is worth mentioning that special attention must be paid to the key classes to keep the entire design of software system. So, classes that have high coupling (CBO, CE, RFC), and lack of cohesion (LCOM) would be considered as important classes to assess the overall software design.

Based on our findings, coupling (CBO, CE, RFC) and lack of cohesion (LCOM) metrics are the most significant predictors of classes' testing coverage, this outcome captures RQ3.

Table 6. Mean of key classes (KC) and non-key classes (NKC) for *R*1.

Metric	Mean	
	KC	NKC
WMC	23.03	9.17
DIT	4.87	1.97
NOC	0.27	1.03
CBO	22.97	16.93
RFC	71.13	57.76
LCOM	34.93	32
Ca	11.37	12.37
Ce	12.5	8.26
NPM	8.1	8.13
LCOM3	0.899293	0.873763
LOC	66.4	60.45
DAM	0.515387	0.715258
MOA	3.8	2.37
MFA	0.19636	0.432754
CAM	0.476053	0.419729
IC	0.433333	0.806543
CBM	1.97	1.55
AMC	40.64784	38.692573
CC	24.5	21.66

4.4. Software testing objectives

Testing coverage is defined as a technique which decides whether the test cases are actually covering the application code and how much code is executed while running the test on software system. Regression testing is the main costly phase in the software testing process since it requires rerunning all test cases in all test suites. This takes a long time especially with large number of test cases. For this reason, focusing on key classes in terms of testing coverage perspective would be helpful in mitigating testing cost. The proposed approach better achieves the software regression testing goals. The proposed approach attempts to help the software engineer to focus on key classes which are (complex, longer methods, larger size of code, high coupling, lack of cohesion, and frequently inherited by other classes). Key classes have higher priority than other classes based on the testing coverage aspect, such classes should be executed earlier in the regression testing cycle. Selecting key coverage classes would decrease the software regression testing cost instead of testing a large number of non-key classes. The proposed approach guides developers to get earlier feedback about software faults which lead to improve the software testing objectives. Consequently, this outcome addresses RQ4.

4.5. Model validation

The proposed approach captures testing coverage as another new method to determine key classes. To verify the validity of the proposed approach, different scopes of software systems are studied with three object-oriented software projects: Apache Ant [62], Apache POI [63], and Maven net. bytebuddy [64]. Table 7 presents statistical description about the multiple projects' dataset.

Apache Ant is a Java library and command-line tool used in both industrial and open-source environments. It is used to build Java applications. Also, Apache POI is an open-source java library used to define and manipulate various file formats based on Microsoft Office. In addition, Byte Buddy is a Java library used for

creating and redefining Java classes at run time, and the Byte Buddy agent enables attaching an agent to the local or a remote virtual machine.

Table 7. The description of multiple projects dataset.

Project	Release	No. of classes	LOC	Mean	Standard Deviation	*Upper-IQR*
Apache Ant	1.7.0	180	81036	0.003705974	0.001784816	0.001882715
	1.9.9	228	78126	0.004905548	0.001883387	0.002943326
Apache POI	3.0.0	334	129232	0.008179169	0.005603231	0.003813686
Maven net.byte buddy	1.5.0	1429	956159	0.002427401	0.001861197	0.001347931

The results indicate the key classes for each investigated software release. For Apache Ant 1.7.0, the key coverage-classes labelled as: *Redirector*, *FilterSet*, *UnknownElement*, *MailMessage*, and *ANTLR* for Apache Ant 1.9.9 has five key classes: *Redirector*, *ModifiedSelector*, *FilterSet*, *UnknownElement*, and *MacroInstance*. Also, the Apache POI release has four key coverage-classes named as: *Workbook*, *AbstractFunctionPtg*, *FormulaParser*, and *BinaryTree*. In addition, the investigated release in Maven net.bytebuddy project has six key coverage-classes defined as: *MethodDescription*, *InstrumentedType*, *ElementMatchers*, *TypeDescription*, *InvokeDynamic*, and *AgentBuilder* as shown in Table 8. Furthermore, after applying multiple linear regression on these projects, the results confirm that the most important predictors of classes' testing coverage are related to complexity, coupling, lack of cohesion, size, and inheritance properties of the classes. Notably, this approach can be applied within the same project or with multiple different projects.

Table 8. Selected classes for the multiple projects.

Project	Release no.	First selected classes	Second selected classes	Highest Ranked class	Lowest Ranked class
Apache Ant	1.7.0	40	5	Redirector	Typedef
	1.9.9	34	5	Redirector	Typedef
Apache POI	3.0.0	23	4	Workbook	OperationPtg
Maven net.bytebuddy	1.5.0	84	6	Method Description	ByteBuddy PrefixInterceptor

In order to compare our results with the related approaches as in Table 9, we consider the approaches that used the same investigated projects as Apache Lucene [65] and Apache Ant [9, 20, 65, 66]. These approaches detected key classes in terms of source code comprehension by using different structural properties (coupling, inheritance, complexity, etc.) of software classes.

However, our approach captures key classes in the perspective of testing coverage and only uses structural properties of classes for the model evaluation. The other studies [9, 20, 65, 66] used either software documentation or the developers feedback to get the ground-truth classes.

The key classes of Apache Ant project automatically obtained from the study [9] whereas the key classes of Apache Lucene project [66] were collected from the project documentation. To deal with limited availability of ground truth classes for validation, an interactive approach will be constructed to guide further testing coverage direction in the future research. To the best of our knowledge, this proposed approach is the first one which employs the testing coverage as a baseline for ranking and selecting key classes in object-oriented software. All the related studies overlooked to focus on testing coverage perspective and considered only the source code perspective. So, our approach mainly pinpoints this critical limitation by implementing a statistical approach which provides a better understanding of software regression testing and evaluating the software design.

Table 9. Comparing with the related approaches.

Project	Approach	No. releases	Avg. no. classes per release	Context of selection
Apache Ant	Sora [20]	1	524	Source code comprehension
	Wang et al. [65]	1	403	Source code comprehension
	Vale and Maia [66]	1	475	Source code comprehension
	Zaidman and Demeyer [9]	1	127	Source code comprehension
	Our approach	2	204	Testing coverage
Apache Lucene	Vale and Maia [66]	1	415	Source code comprehension
	Our approach	23	3321	Testing coverage
Apache POI	Our approach	1	334	Testing coverage
Maven net.bytebuddy	Our approach	1	1429	Testing coverage

The related approaches select key classes based on program comprehension while our approach selects key classes based on testing coverage. Our approach sheds light on the need of considering key classes while designing the corresponding test cases. For example, among all ground-truth classes of Apache Lucene project such in the studies [9, 66], two classes (*IndexWriter, QueryParser*) were found as key coverage classes in our approach. The remaining key classes attain considerably low levels of testing-coverage due to the limitations of the test cases comprehensiveness. In Apache Ant project, the *UnknownElement* class is considered as key class and key coverage-class in both contexts. With respect to the related dataset of the previous approaches, our approach introduces the top 30 key coverage-classes in Apache Lucene project and the top five key coverage-classes in Apache Ant project. Based on the context of our approach, we would recommend our findings as guidelines for future studies that will explore the vital role of testing coverage in selecting key classes for software system.

4.6. Threats to validity

This study assumes that all the classes selected in this approach were the key coverage classes. There is a possibility that some of these classes were not important or not the key classes of the software. In other words, there are other important classes which have been missed by this proposed approach since it considers only the testing coverage perspective. Feedback from the software

developers may improve the accuracy of these key classes. Nevertheless, obtaining the feedback needs more effort.

However, this study lacks any ground-truth for key classes in terms of testing coverage. The precision and recall measures for the most related studies [9, 20, 65, 66] were calculated based on the number of the ground-truth classes.

5. Conclusions and future work

This paper proposes a statistical based approach which employs testing coverage information to rank and select key coverage-classes in object-oriented software. The proposed approach consists of three statistical phases including (1) Ranking all the classes of the software system by using the sequential ranking method, (2) Selecting the highly coverage classes by utilizing the box blot rule, and (3) Selecting the key classes by applying Chebyshev's theorem based on the normality of the classes coverage distribution. Experiments are conducted on 23 successive releases of large open-source software system, Apache Lucene. Also, four extra releases of three different projects (Apache Ant, Apache POI, and Maven net. bytebuddy) are extended to validate the proposed approach.

To evaluate the proposed approach, multiple linear regression is applied to show the relationship between testing coverage and structural properties of classes. In conclusion, the results have shown that there is a statistically significant correlation between testing coverage and the structural properties of classes in terms of complexity, coupling, lack of cohesion, size, and inheritance. Our approach helps software engineers to assess software design and better achieve software testing objectives by highlighting key classes at the earlier cycles of regression testing. The proposed approach is expected to use of testing coverage metrics to improve the software quality assurance and maintainability.

As future work, we plan to get feedback from software developers to obtain the ground-truth classes. We intend to extend this approach to include the program comprehension context, this will facilitate the comparison within the same scope of the previous studies. Another future direction is to utilize different software projects with different sizes. Also, we plan to develop a tool that will automate the proposed approach.

Nomenclatures

C	Classes set
c_i	i^{th} class
Cov^B	Class test coverage, Branch perspective
Cov^C	Class test coverage, Cyclomatic complexity perspective
Cov^I	Class test coverage, Instruction perspective
Cov^L	Class test coverage, Line perspective
Cov^M	Class test coverage, Method perspective
JCW_{ci}	JaCoCo coverage weight of class c_i (a single class weight)
JCW_{ri}	JaCoCo coverage weight of release r_i (weight of all classes in r_i)
JCW_{S1}^{ri}	Selecting of highly coverage classes
JCW_{S2}^{ri}	Selecting of the key classes
R	Releases set

r_i	i^{th} release
U_IQR_{ri}	Upper inter quartile range of release r_i

Greek Symbols

μ μ	Arithmetic mean
σ σ	Arithmetic standard deviation

Abbreviations

AMC	Average Method Complexity
CA	Afferent Couplings
CAM	Cohesion Among Methods
CBM	Coupling Between Methods
CBO	Coupling Between Object Classes
CC	Mccabe's Cyclomatic Complexity
CE	Efferent Couplings
DAM	Data Access Metric
DIT	Depth of Inheritance Tree
IC	Inheritance Coupling
JCW	Jacoco Coverage Weight
KC	Key Classes
LCOM	Lack of Cohesion in Methods
LOC	Lines of Code
MFA	Measure of Functional Abstraction
MOA	Measure of Aggregation
NKC	Non-key classes
NOC	Number of Children
NPM	Number of Public Methods
RFC	Response for a Class
SQA	Software Quality Assurance
WMC	Weighted Method Per Class

References

1. Magalhães, C.; Andrade, J.; Perrusi, L.; Mota, A.; Barros, F.; and Maia, E. (2020). HSP: A hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study. *Journal of Systems and Software*, 159, 110430.
2. Coviello, C.; Romano, S.; Scanniello, G.; Marchetto, A.; Corazza, A.; and Antoniol, G. (2020). Adequate vs. inadequate test suite reduction approaches. *Information and Software Technology*, 119, 106224.
3. Runeson, P. (2006). A survey of unit testing practices. *IEEE* 25*th International Symposium on Software Reliability Engineering*, 23(4), 22-29.
4. Grambow, G.; Oberhauser, R.; and Reichert, M. (2011). Contextual injection of quality measures into software engineering processes. *International Journal on Advances in Software*, 4(2), 76-99.
5. Fernández, P.O.; David, M.; Duma, D.C.; Ronchieri, E.; Gomes, J.; and Salomoni, D. (2020). Software quality assurance in indigo-data cloud project: A converging evolution of software engineering practices to support European research e-infrastructures. *Journal of Grid Computing*, 18, 81-98.

6. Mahapatra, S.; and Mishra, S. (2020). Automated software engineering: A deep learning-based approach. Chapter: Usage of machine learning in software testing. *Learning and Analytics in Intelligent Systems.* Springer, 39-54.
7. Walia, M.; Gupta, A.; and Singla, R.K. (2017). Improvement in key project performance indicators through deployment of a comprehensive test metrics advisory tool. *International Journal of Advanced Research in Computer Science*, 8(5), 1236-1241.
8. Shatnawi, R.; and Li, W. (2011). An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications*, 5(4), 127-149.
9. Zaidman, A.; and Demeyer, S. (2008). Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6), 387-417.
10. Ducasse, S.; and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4), 573-591.
11. Bethea, R.M. (2018). *Statistical methods for engineers and scientists* (3rd ed.). USA: CRC Press.
12. Ratner, B. (2017). *Statistical and machine-learning data mining: techniques for better predictive modeling and analysis of big data* (3rd ed.). Florida: CRC Press.
13. Jaggia, S.; Kelly, A.; Salzman, S.; Olaru, D.; Sriananthakumar, S.; Beg, R.; and Leighton, C. (2016). *Essentials of Business Statistics: Communicating with numbers*. Australia: McGrawhill Education.
14. Zaidman, A.; Calders, T.; Demeyer, S.; and Paredaens, J. (2005). Applying webmining techniques to execution traces to support the program comprehension process. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering.* Manchester, UK, 134-142.
15. Zimmermann, T.; Zeller, A.; Weissgerber, P.; and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6), 429-445.
16. Wang, M.C.; and Pan, W.F. (2012). A comparative study of network centrality metrics in identifying key classes in software. *Journal of Computational Information Systems*, 8(24), 10205-10212.
17. Ding, Y.; Li, B.; and He, P. (2016). An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering*, 2016.
18. Meyer, P.; Siy, H.; and Bhowmick, S. (2014). Identifying important classes of large software systems through k-core decomposition. *Advances in Complex Systems*, 17(07n08), 1550004.
19. Hammad, M.; Collard, M.L.; and Maletic, J.I. (2010). Measuring class importance in the context of design evolution. *Proceedings of the IEEE* 18*th International Conference on Program Comprehension*. Minho, Braga, 148-151.
20. Şora, I. (2015). A PageRank based recommender system for identifying key classes in software systems. *Proceedings of the* 2015 *IEEE* 10*th Jubilee International Symposium on Applied Computational Intelligence and Informatics.* Timisoara, Romania, 495-500.

21. Osman, M.H.; Chaudron, M.R.; and Putten, P.V.D. (2013). An analysis of machine learning algorithms for condensing reverse engineered class diagrams. *Proceedings of the* 2013 *IEEE International Conference on Software Maintenance.* Massachusetts, United States, 140-149.
22. Thung, F.; Lo, D.; Osman, M.H.; and Chaudron, M.R. (2014). Condensing class diagrams by analyzing design and network metrics using optimistic classification. *Proceedings of the* 22*nd International Conference on Program Comprehension.* Hyderabad, India, 110-121.
23. Sager, T.; Bernstein, A.; Pinzger, M.; and Kiefer, C. (2006). Detecting similar Java classes using tree algorithms. *Proceedings of the* 2006 *international workshop on Mining software repositories.* Shanghai, China, 65-71.
24. Khan, R.A.; and Mustafa, K. (2009). Metric based testability model for object oriented design (MTMOOD). *ACM SIGSOFT Software Engineering Notes*, 34(2), 1-6.
25. Kout, A.; Toure, F.; and Badri, M. (2011). An empirical analysis of a testability model for object-oriented programs. *ACM SIGSOFT Software Engineering Notes*, 36(4), 1-5.
26. Pinzger, M.; Gall, H.; Fischer, M.; and Lanza, M. (2005). Visualizing multiple evolution metrics. *Proceedings of the* 2005 *ACM symposium on Software visualization.* St. Louis, Missouri, 67-75.
27. Robillard, M. P. (2005). Automatic generation of suggestions for program investigation. *Proceedings of the* 10*th European software engineering conference held jointly with* 13*th ACM SIGSOFT international symposium on Foundations of software engineering.* Lisbon, Portugal, 11-20.
28. Bieman, J.M.; Andrews, A.A.; and Yang, H.J. (2003). Understanding change-proneness in OO software through visualization. *Proceedings of the* 11*th IEEE International Workshop on Program Comprehension.* Massachusetts, United States, 44-53.
29. Hoffmann, M.R. (2014). JaCoCo Java code coverage library. Retrieved December 5, 2019, from https://www.jacoco.org/jacoco/.
30. Hoffmann, M.R.; Janiczak, B.; and Mandrikov, E. (2011). Eclemma-jacoco java code coverage library. Retrieved January 5, 2020, from http://eclemma.org/jacoco/.
31. Hoffmann, M.; Mandrikov, E.; and Friedenhagen, M. (2020). JaCoCo java code coverage library. Retrieved January 5, 2020, from https://www.eclemma.org/jacoco/.
32. Horváth, F.; Gergely, T.; Beszédes, Á.; Tengeri, D.; Balogh, G.; and Gyimóthy, T. (2019). Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal*, 27(1), 79-123.
33. Noemmer, R.; and Haas, R. (2020). An evaluation of test suite minimization techniques. *Proceedings of the* 12*th International Conference on Software Quality.* Vienna, Austria, 51-66.
34. Afifi, A.; May, S.; Donatello, R.; and Clark, V.A. (2019). *Practical multivariate analysis*. USA: CRC Press.
35. van Deursen, A.; Aniche, M.; Boone, C.; Cunha, M.L.; and Nadeem, A. (2019). *Software quality and testing: SQT labwork, CSE*1110 (2018/2019 *ed.*). Delft University of Technology.

36. Chioteli, E.; Batas, I.; and Spinellis, D. (in press). Does unit-tested code crash? A case study of eclipse. *Proceedings of the* 2020 *Online Mining Software Repositories Conference*.
37. Grano, G.; Titov, T. V.; Panichella, S.; and Gall, H. C. (2019). Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process*, 31(9), e2158.
38. Virgínio, T.; Santana, R.; Martins, L.A.; Soares, L. R.; Costa, H.; and Machado, I. (2019). On the influence of test smells on test coverage. *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. Salvador, Brazil, 467-471.
39. Kifetew, F.; Devroey, X.; and Rueda, U. (2019). Java unit testing tool competition-seventh round. *Proceedings of the* 2019 *IEEE/ACM* 12*th International Workshop on Search-Based Software Testing* (*SBST*). Montreal Quebec, Canada, 15-20.
40. Carlson, R.; Do, H.; and Denton, A. (2011). A clustering approach to improving test case prioritization: An industrial case study. *Proceedings of the* 27*th IEEE International Conference on Software Maintenance* (*ICSM*). Williamsburg, USA, 382-391.
41. Kalaimagal, R.; and Jacob, S.G. (2015). Improved random forest algorithm for software defect prediction through data mining techniques. *International Journal of Computer Applications*, 117(23), 18-22.
42. Paramshetti, P.; and Phalk, D. (2015). Software defect prediction for quality improvement using hybrid approach. *International Journal of Application or Innovation in Engineering & Management,* 4(6), 099-104.
43. Rakić, G.; Tóth, M.; and Budimac, Z. (2020). Toward recursion aware complexity metrics. *Information and Software Technology*, 118, 106203.
44. Lenhard, J.; Blom, M.; and Herold, S. (2019). Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Software Quality Journal*, 27(1), 241-274.
45. Siavvas, M.G.; Chatzidimitriou, K.C.; and Symeonidis, A.L. (2017). QATCH-An adaptive framework for software product quality assessment. *Expert Systems with Applications*, 86, 350-366.
46. Saifan, A.A.; and Al Smadi, N. (2019). Source code-based defect prediction using deep learning and transfer learning. *Intelligent Data Analysis*, 23(6), 1243-1269.
47. Scitools (2018). Scitool. Retrieved January 5, 2020, from http://www.scitools.com/.
48. Spinellis, D.D. (2009). Ckjm-a tool for calculating chidamber and kemerer java metrics. Retrieved February 10, 2020, from https://www.spinellis.gr/sw/ckjm/.
49. Chidamber, S.R.; and Kemerer, C.F. (1991). Towards a metrics suite for object oriented design. *Proceedings of the Object-oriented programming systems, languages, and applications conference.* Phoenix, USA, 197-211.
50. Henderson-Sellers, B. (1995). *Object-oriented metrics: measures of complexity*. New Jersey: Prentice-Hall, Inc.
51. Bansiya, J.; and Davis, C.G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1), 4-17.

52. Tang, M.H.; Kao, M.H.; and Chen, M.H. (1999). An empirical study on object-oriented metrics. *Proceedings of the sixth international software metrics symposium* (*Cat. No. PR*00403). Boca Raton, USA 242-249.
53. Martin, R. (1994). OO design quality metrics. *An analysis of dependencies*, 12(1), 151-170.
54. Yi, T.; and Fang, C. (2018). A complexity metric for object-oriented software. *International Journal of Computers and Applications*, 42(6), 544-549.
55. Usman, M.; Britto, R.; Damm, L.O.; and Börstler, J. (2018). Effort estimation in large-scale software development: An industrial case study. *Information and Software technology*, 99, 21-40.
56. Azzopardi, L.; Moshfeghi, Y.; Halvey, M.; Alkhawaldeh, R.S.; Balog, K.; Di Buccio, E.; and Palchowdhury, S. (2017). Lucene4IR: Developing information retrieval evaluation resources using Lucene. *Proceedings of the ACM SIGIR Forum,* 50(2), 58-75.
57. Lambert, L. (2016). *Apache solr essentials*. North Charleston: CreateSpace Independent Publishing Platform.
58. Antinyan, V.; Derehag, J.; Sandberg, A.; and Staron, M. (2018). Mythical unit test coverage. 2019 *IEEE/ACM* 41*st International Conference on Software Engineering: Software Engineering in Practice* (*ICSE-SEIP*). Montreal, Canada, 73-79.
59. Fifo, M.; Enoiu, E.; and Afzal, W. (2019). On measuring combinatorial coverage of manually created test cases for industrial software. *Proceedings of the* 2019 *IEEE International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*). Xi'an, China, 264-267.
60. Ammann, P.; and Offutt, J. (2016). *Introduction to software testing*. UK: Cambridge University Press.
61. Malhotra, R. (2016). *Empirical research in software engineering: concepts, analysis, and applications* (1st ed.). New York: CRC Press.
62. Apache Ant. (2010). The Apache ant project. Retrieved February 3, 2020, from https://ant.apache.org/.
63. Oliver, A.C.; Stampoultzis, G.; Sengupta, A.; Klute, R.; and Fisher, D. (2002). Apache POI-the Java API for Microsoft documents. Retrieved March 7, 2020, from https://poi.apache.org/.
64. Winterhalter, R. (2017). Byte buddy. Retrieved March 15, 2020. from https://mvnrepository.com/artifact/net.bytebuddy.
65. Wang, J.; Ai, J.; Yang, Y.; and Su, W. (2017). Identifying key classes of object-oriented software based on software complex network. *Proceedings of the* 2017 2*nd International Conference on System Reliability and Safety* (*ICSRS*). Milan, Italy, 444-449.
66. Vale, L.D.N.; and Maia; M.D.A (2019). Key classes in object-oriented systems: Detection and assessment. *International Journal of Software Engineering and Knowledge Engineering*, 29(10), 1439-1463.