

Logical dependencies: extraction from the versioning system and usage^{*}

Adelina Stana¹ and Ioana Sora²

¹ Stana Adelina Diana

Address

stana.adelina.diana@gmail.com

² Sora Ioana

Address

ioana.sora@cs.upt.ro

Abstract. The abstract goes here, and keywords in a new paragraph.

Keywords: logica ldependencies, versioning system, key classes.

1. Introduction

The goal of the thesis is to develop methods for analyzing legacy software systems by using historical information extracted from the versioning systems.

We divided our work into two main parts, the first part covers the historical information collection and filtering and was presented in the first report. And the second part covers the usage of the collected information and will be presented in this second report.

In the first report, we presented the filtering process of co-changing pairs into logical dependencies. The co-changing pairs are pairs of classes that record co-changes in the versioning system. The logical dependencies are those co-changing pairs that remain after filtering. We filter the co-changing pairs to increase their veridicality and to decrease the size of the extracted information. Based on the results obtained in the first report, we decided to work with the following filters: the filter based on commit size (cs) and the filter based on connection strength. With the filter based on commit size, we filter out each commit transaction with more than 10 files changed. Big commit transactions (more than 10 files) can be related to refactoring of names, spellchecks, or file reformatting and not to actual code changes. This filter will reduce the amount of co-changing pairs extracted.

With the filter based on connection strength, we filter out co-changing pairs that are not strongly connected in the versioning system. For this filter, we did not previously establish a hard threshold value because we want to check how different values of the threshold will impact the results obtained after using the extracted logical dependencies.

This second report mainly focuses on using the logical dependencies extracted in identifying key (important) classes in a software system. The identification of key classes was previously researched solely by using software dependencies. Our approach is to take previously researched software systems and the results obtained based on those systems as a baseline. In addition to the baseline approach of using only structural dependencies, we

^{*} If this is an extended version of a conference paper, it should be clearly stated here.

1 will add the logical dependencies extracted, export the results, and compare them with the
2 baseline results.

3 Additionally, in this report, we will compare the number of logical dependencies with
4 other software metrics. With this comparison, we mainly want to observe if the number of
5 logical dependencies can be connected to other metrics or if is complementary with other
6 metrics.

7 The report is structured in two main chapters, in chapter ?? is presented the state-of-
8 the-art in key classes detection, how the results obtained are evaluated, and the baseline
9 approach for our research. In chapter ?? is presented the key classes detection by using
10 logical dependencies and the evaluation of the new results obtained.

11 2. The concept of key classes

12 Zaidman et al [21] were the first to introduce the concept of key classes and it refers to
13 classes that can be found in documents written to provide an architectural overview of
14 the system or an introduction to the system structure. Tahvildari and Kontogiannis have a
15 more detailed definition regarding key classes concept: “Usually, the most important con-
16 cepts of a system are implemented by very few key classes which can be characterized
17 by the specific properties. These classes, which we refer to as key classes, manage many
18 other classes or use them in order to implement their functionality. The key classes are
19 tightly coupled with other parts of the system. Additionally, they tend to be rather com-
20 plex, since they implement much of the legacy system’s functionality” [17]. Also, other
21 researchers use a similar concept as the one defined by Zaidman but under different terms
22 like important classes [8] or central software classes [16].

23 The key class identification can be done by using different algorithms with different
24 inputs. In the research of Osman et al., the key class identification is made by using a
25 machine learning algorithm and class diagrams as input for the algorithm [11]. Thung
26 et al. builds on top of Osman et al.’s approach and adds network metrics and optimistic
27 classification in order to detect key classes [18].

28 Zaidman et al. use a webmining algorithm and dynamic analysis of the source code to
29 identify the key classes [21].

30 Sora et al. use a page ranking algorithm for finding key classes and static analysis of
31 the source code [2], [9], [3], [14]. In [10] the authors use in addition to the previous re-
32 search also other class attributes to identify important classes. The page ranking algorithm
33 is a customization of PageRank, the algorithm used to rank web pages [12]. The PageR-
34 ank algorithm works based on a recommendation system. If one node has a connection
35 with another node, then it recommends the second node. In previous works, connections
36 are established based on structural dependencies extracted from static code analysis. If A
37 has a structural dependency with B, then A recommends B, and also B recommends A.

38 The ranking algorithm ranks all the classes from the source code of the system ana-
39 lyzed according to their importance. To identify the important classes from the rest of the
40 classes a threshold for TOP classes from the top of the ranking is set. The TOP threshold
41 value can go from 1 to the total number of classes found in the system.

42 Some researchers [21], [4], [13] consider that 15% of the total number of classes of
43 the system is a suited value for the TOP threshold. Other researchers [10] consider that

1 15% of the total number of classes is a too high value for the TOP threshold and suggest
 2 that a value in the range of 20–30 is better.

3 3. Metrics for results evaluation

4 To evaluate the quality of the key classes ranking algorithm and solution produced, the
 5 key classes found by the algorithm are compared with a reference solution.

6 The reference solution is extracted from the developer documentation. Classes men-
 7 tioned in the documentation are considered key classes and form the reference solution
 8 (ground truth) used for validation [19].

9 For the comparison between both solutions, is used a classification model. The quality
 10 of the solution produced is evaluated by using metrics that evaluate the performance of
 11 the classification model, such as Precision-Recall and Receiver Operating Characteristic
 12 Area Under Curve (ROC-AUC).

13 A classification model (or "classifier") is a mapping between expected results and
 14 predicted results [6], [1]. Both results can be labeled as positive or negative, which leads
 15 us to the confusion matrix from figure 1.

Expected Result \ Predicted Result	Positive	Negative
Positive	<i>True Positive</i>	<i>False Positive</i>
Negative	<i>False Negative</i>	<i>True Negative</i>

Fig. 1. Confusion matrix

16 The confusion matrix has the following outcomes:

- 17 – *true positive*, if the expected result is positive and the predicted result is also positive.
- 18 – *false positive*, if the expected result is positive but the predicted result is negative.
- 19 – *false negative*, if the expected result is negative but the predicted result is positive.
- 20 – *true negative*, if the expected result is negative and the predicted result is also nega-
 21 tive.

22 *Precision-recall*

23 Precision is the ratio of True Positives to all the positives of the result set.

$$precision = \frac{TP}{TP + FN} \quad (1)$$

24 The recall is the ratio of True Positives to all the positives of the reference set.

$$recall = \frac{TP}{TP + FP} \quad (2)$$

As mentioned in section 2, to distinguish the key classes from the rest of the classes a TOP threshold is used. Some researchers consider that 15% of the total classes is the best value for the TOP threshold and others consider that the value should be in the range of 20-30.

The precision-recall metric is suited if the threshold value is fixed. If the threshold value is variable, then metrics that capture the behavior over all possible values must be used. Such metric is the Receiver Operating Characteristic metric.

Receiver Operating Characteristic Area Under Curve

The ROC graph is a two-dimensional graph that has on the X-axis plotted the false positive rate and on the Y-axis the true positive rate. By plotting the true positive rate and the false positive rate at thresholds that vary between a minimum and a maximum possible value we obtain the ROC curve. The area under the ROC curve is called Area Under the Curve (AUC).

The true positive rate of a classifier is calculated as the division between the number of true positive results identified and all the positive results identified:

$$True\ positive\ rate(TPR) = \frac{TP}{TP + FN} \quad (3)$$

The false positive rate of a classifier is calculated as the division between the number of false positive results identified and all the negative results identified:

$$False\ positive\ rate(FPR) = \frac{FP}{FP + TN} \quad (4)$$

In multiple related works, the ROC-AUC metric has been used to evaluate the results for finding key classes of software systems. For a classifier to be considered good, its ROC-AUC metric value should be as close to 1 as possible, when the value is 1 then the classifier is considered to be perfect.

Osman et al. obtained in their research an average Area Under the Receiver Operating Characteristic Curve (ROC-AUC) score of 0.750 [11]. Thung et al. obtained an average ROC-AUC score of 0.825 [18] and Sora et al. obtained an average ROC-AUC score of 0.894 [10].

4. Baseline approach

We use the research of I. Sora et al [10] as a baseline for our research involving the usage of logical dependencies to find key classes. The baseline approach uses a tool that takes as an input the source code of the system and applies ranking strategies to rank the classes according to their importance.

In order to rank the classes according to their importance, different class metrics are used [4], [21], [13]. Below are presented some of the class metrics used in the baseline approach in order to rank the classes according to their importance.

1 **Class attributes that characterize key classes** The metrics used in the baseline research
 2 can be grouped into the following categories:

- 3 – class size metrics: number of fields (NoF), number of methods (NoM), global size
 4 (Size = NoF+NoM).
- 5 – class connection metrics, any structural dependency between two classes:
 - 6 • CONN-IN, the number of distinct classes that use a class;
 - 7 • CONN-OUT, the total number of distinct classes that are used by a class;
 - 8 • CONN-TOTAL, the total number of distinct classes that a class uses or are used
 9 by a class (CONN-IN + CONN-OUT).
 - 10 • CONN-IN-W, the total weight of distinct classes that use a class.
 - 11 • CONN-OUT-W, the total weight of distinct classes that are used by a class.
 - 12 • CONN-TOTAL-W, the total weight of all connections of the class (CONN-IN-W
 13 + CONN-OUT-W) [10].
- 14 – class pagerank values, previous research use pagerank values computed on both di-
 15 rected and undirected, weighted and unweighted graphs:
 - 16 • PR - value computed on the directed and unweighted graph;
 - 17 • PR-W - value computed on the directed and weighted graph;
 - 18 • PR-U - value computed on the undirected and unweighted graph;
 - 19 • PR-U-W - value computed on the undirected and weighted graph;
 - 20 • PR-U2-W - value computed on the weighted graph with back-recommendations
 21 [2], [9], [10], [14].

22 Based on the class attributes presented, all the classes of the system are ranked. To
 23 differentiate the important (key) classes from the rest of the classes, a TOP threshold for
 24 the top classes found is set. The threshold vary between 20 and 30 classes.

25 The baseline approach not only identifies the key classes but also evaluates the perfor-
 26 mance of the solution produced. The same approach as the one presented in section 3 is
 27 used for the evaluation of the results. The key classes found by the ranking algorithm are
 28 compared with a reference solution that is extracted from the developer documentation by
 29 using a classification model.

30 The true positives (TP) are the classes found in the reference solution and also in the
 31 top TOP ranked classes. False positives (FP) are the classes that are not in the reference
 32 solution but are in the TOP ranked classes. True Negatives (TN) are classes that are found
 33 neither in the reference solution nor in the TOP ranked classes. False Negatives (FN) are
 34 classes that are found in the reference solution but not found in the TOP ranked classes.

35 Due to the fact that the TOP threshold is varied, the Receiver Operating Characteristic
 36 Area Under Curve metric is used for the evaluation of the results.

37 The entire workflow of the baseline approach that was presented above is also pre-
 38 sented in figure 2.

39 5. Data set used

40 In this section, we will look over all the systems studied in the baseline research presented
 41 in section 4, and we will try to identify the systems that could be used also in our current
 42 research involving logical dependencies.

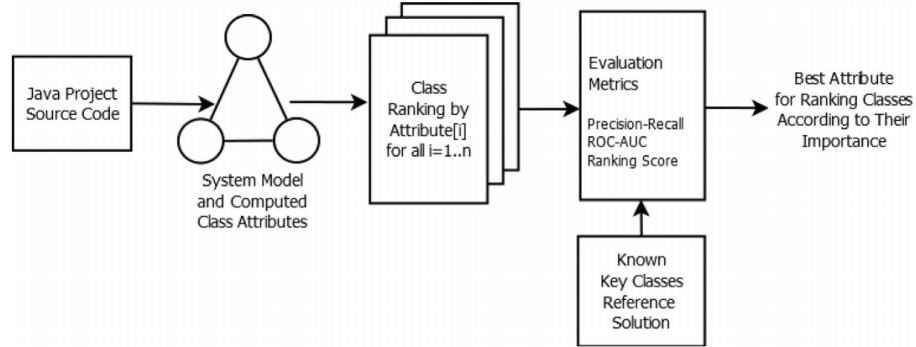


Fig. 2. Overview of the baseline approach. Reprinted from “Finding key classes in object-oriented software systems by techniques based on static analysis.” by Ioana Sora and Ciprian-Bogdan Chirila, 2019, Information and Software Technology, 116:106176. Reprinted with permission.

1 The research of I. Sora et al [10] takes into consideration structural public dependen-
 2 cies that are extracted using static analysis techniques and was performed on the object-
 3 oriented systems presented in table 1.

4 The requirements for a system to qualify as suited for investigations using logical
 5 dependencies are: has to be on GitHub, has to have release tags to identify the version,
 6 and also has to have an increased number of commits. From the total of 14 object-oriented
 7 systems listed in the paper [10], 13 of them have repositories in Github 2. And from the
 8 found repositories we identified only 6 repositories that have the same release tag as the
 9 specified version from table 1. It is important to identify the correct release tag for each
 10 repository to limit the commits further analyzed by date. Only commits that were made
 11 until the specified release are considered and analyzed. The commits number found on the
 12 remaining 6 repositories varies from 19108 commits for Tomcat Catalina to 149 commits
 13 for JHotDraw. In order to have more accurate results, we need a significant number of
 14 commits, so we reached the conclusion that only 3 systems can be used for key classes
 15 detection using logical dependencies: Apache Ant, Hibernate, and Tomcat Catalina. From
 16 all the systems mentioned in table 1 Apache Ant is the most used and analyzed in other
 17 works [15], [5], [20], [7].

18 6. Measurements using logical dependencies

19 As we mentioned in the beginning the purpose is to check if the logical dependencies can
 20 improve key class detection.

21 As presented in section 4, and section 2 the key class detection was done by using
 22 structural dependencies of the system. In this section, we will use the same tool used in
 23 the baseline approach presented in section 4, and we will add a new input to it, the logical
 24 dependencies.

25 Below is a comparison between the new approach and baseline approach, how we
 26 collect the logical dependencies, the results obtained previously, and the new results ob-

Table 1. Analyzed software systems in previous research paper.

ID	System	Description
S1	Apache Ant	Java library and command line tool that drive the build processes as targets and extension points depending
S2	Argo UML	UML modelling tool with support for all UML diagrams.
S3	GWT Portlets	Open source web framework for building GWT (Google Web Toolkit) Applications.
S4	Hibernate	Persistence framework for Java.
S5	javaclient	Java distributed application for playing with robots
S6	jEdit	Java mature text editor for programmers.
S7	JGAP	Genetic Algorithms and Genetic Programming Java library.
S8	JHotDraw	JHotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Java.
S9	JMeter	JMeter is a Java application designed to load test functional behavior and measure performance
S10	Log4j	Logging Service
S11	Mars	The Mars Simulation Project is a Java project that models and simulates human settlements on Mars planet
S12	Maze	The Maze-solver project simulates an artificial intelligence algorithm on a maze
S13	Neuroph	Neuroph is a Java neural network framework.
S14	Tomcat Catalina	The Apache Tomcat project is an open-source implementation of JavaServlet and JavaServerPages technol
S15	Wro4J	The Wro4J is a web resource (JS and CSS) optimizer for Java.

Table 2. Found systems and versions of the systems in GitHub.

ID	System	Version	Release Tag name	Commits number
S1	Apache Ant	1.6.1	rel/1.6.1	6713
S2	Argo UML	0.9.5	not found	0
S3	GWT Portlets	0.9.5 beta	not found	0
S4	Hibernate	5.2.12	5.2.12	6733
S5	javaclient	2.0.0	not found	0
S6	jEdit	5.1.0	not found	0
S7	JGAP	3.6.3	not found	0
S8	JHotDraw	6.0b.1	not found	149
S9	JMeter	2.0.1	v2_1_1	2506
S10	Log4j	2.10.0	v1_2_10-recalled	634
S11	Mars	3.06.0	not found	0
S12	Maze	1.0.0	not found	0
S13	Neuroph	2.2.0	not found	0
S14	Tomcat Catalina	9.0.4	9.0.4	19108
S15	Wro4J	1.6.3	v1.6.3	2871

1 tained. The new results are separated into two categories, the results obtained by using
 2 structural and logical dependencies and the results obtained by using only logical depen-
 3 dencies.

4 **6.1. Comparison with the baseline approach**

5 The baseline approach uses a tool that takes as input the source code of the system to
 6 identify the key classes and the reference solution to evaluate the quality of the solution.
 7 We modified the tool such that it can also take as input the logical dependencies.

8 In order to rank the classes according to their importance, the tool uses different class
 9 metrics. The list of the metrics used in the baseline approach is presented in section 4.
 10 The difference in the metrics used compared with the baseline approach is that we use
 11 a subset of those metrics. The reason why we are not using all the metrics is that the
 12 extracted logical dependencies are undirected. The metrics used by the current approach
 13 are CONN-TOTAL, CONN-TOTAL-W, PR-U, PR-U-W, and PR-U2-W.

14 We did not change the rest of the workflow of the tool. Meaning that the TOP threshold
 15 is varied between 20 and 30 and the resulting solution is evaluated by using the ROC-AUC
 16 metric. The goal being a ROC-AUC (Receiver Operating Characteristic - Area Under the
 17 Curve) metric value as close to 1 as possible.

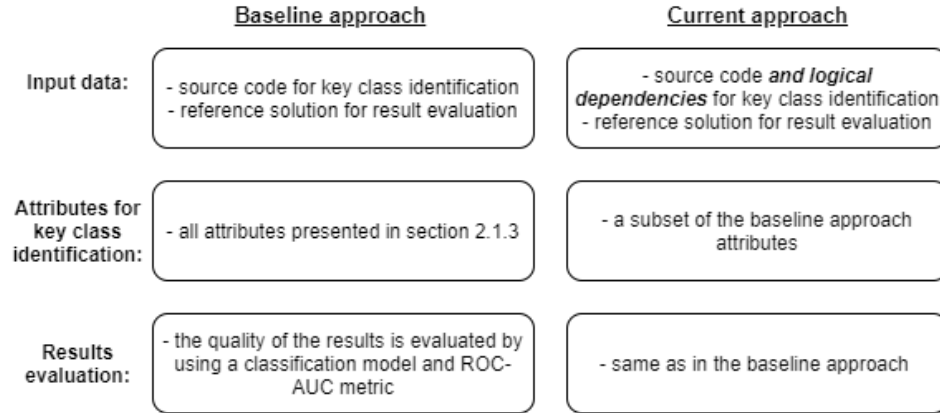


Fig. 3. Comparison between the new approach and the baseline

18 **6.2. Logical dependencies collection and current workflow used**

19 The logical dependencies are those co-changing pairs extracted from the versioning sys-
 20 tem history that remain after filtering. The filtering part consists of applying two filters:
 21 the filter based on commit size and the filter based on connection strength.

22 To determine the connection strength of a pair, we first need to calculate the con-
 23 nection factors for both entities that form a co-changing pair. Assuming that we have a
 24 co-changing pair formed by entities A and B, the connection factor of entity A with entity

- 1 B is the percentage from the total commits involving A that contains entity B. The con-
 2 nection factor of entity B with entity A is the percentage from the total commits involving
 3 B that contain also entity A.

$$\text{connection factor for } A = \frac{100 * \text{commits involving } A \text{ and } B}{\text{total nr of commits involving } A} \quad (5)$$

$$\text{connection factor for } B = \frac{100 * \text{commits involving } A \text{ and } B}{\text{total nr of commits involving } B} \quad (6)$$

- 4 We calculated the connection factor for each entity involved in a co-changing pair and
 5 filtered the co-changing pairs based on it. The rule set is that both entities had to have a
 6 connection factor with each other greater than the threshold value.

- 7 After the filtering part, the remaining co-changing pairs, now called logical dependen-
 8 cies, are exported in CSV files.

- 9 The entire process of extracting co-changing pairs from the versioning system, filter
 10 them, and export the remaining ones into CSV files is done with a tool written in Python.

- 11 The next step is to use the exported logical dependencies for key classes detection. In
 12 order to do that we used the same key class detection tool used in the previous research
 13 presented in section 4. We adapted the tool to be able to process also logical dependencies
 14 because previously the tool used only structural dependencies extracted from the source
 15 code of the software systems. The workflow is presented in figure 4

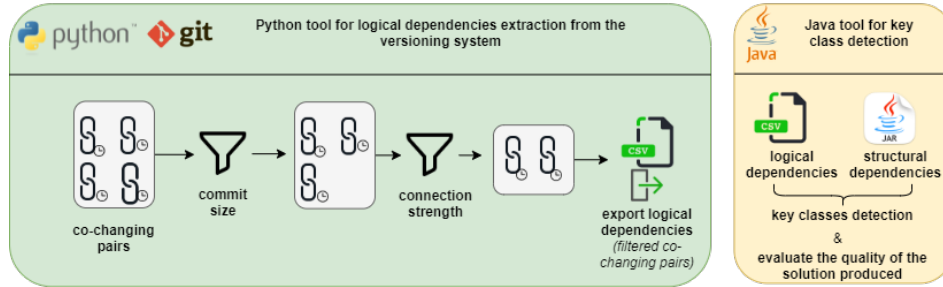


Fig. 4. Workflow for key classes detection

16 6.3. Measurements using only the baseline approach

- 17 In table 3 are presented the ROC-AUC values for different attributes computed for the
 18 systems Ant, Tomcat Catalina, and Hibernate by using the baseline approach. We intend
 19 to compare these values with the new values obtained by using also logical dependencies
 20 in key class detection.

21 6.4. Measurements using combined structural and logical dependencies

- 22 The tool used in the baseline approach runs a graph-ranking algorithm. The graph used
 23 contains the structural dependencies extracted from static source code analysis. Each edge

Table 3. ROC-AUC metric values extracted.

Metrics	Ant	Tomcat	Catalina	Hibernate
PR_U2_W	0.95823	0.92341	0.95823	
PR	0.94944	0.92670	0.94944	
PR_U	0.95060	0.93220	0.95060	
CONN_TOTAL_W	0.94437	0.92595	0.94437	
CONN_TOTAL	0.94630	0.93903	0.94630	

1 in the graph represents a dependency, the entities that form a structural dependency are
 2 represented as vertices in the graph. As mentioned in section 6.1, we modified the tool to
 3 read also logical dependencies and add them to the graph. In this section, we add in the
 4 graph the logical dependencies together with the structural dependencies.

5 In tables 4, 5, and 6, on each line, we have the metric that is calculated and on each
 6 column, we have the connection strength threshold that was applied to the logical de-
 7 pendencies used in identifying the key classes. We started with logical dependencies that
 8 have a connection strength greater than 10%, which means that in at least 10% of the com-
 9 mits involving A or B, A and B update together. Then we increased the threshold value
 10 by 10 until we remained only with entities that update in all the commits together. The
 11 last column contains the results obtained previously by the tool by only using structural
 12 dependencies.

13 As for the new results obtained by combining structural and logical dependencies,
 14 highlighted with orange are the values that are close to the previously registered values but
 15 did not surpass them. Highlighted with green are values that are better than the previously
 16 registered values. At this step, we can also observe that for all three systems measured in
 17 tables 4, 5, and 6, the best values obtained are for connection strength between 40-70%.

Table 4. Measurements for Ant using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.924	0.925	0.926	0.927	0.927	0.927	0.929	0.928	0.928	0.928	0.929
PR	0.914	0.854	0.851	0.866	0.876	0.882	0.887	0.854	0.852	0.852	0.855
PR_U	0.910	0.930	0.933	0.933	0.935	0.934	0.939	0.933	0.933	0.933	0.933
CON_T_W	0.924	0.928	0.931	0.932	0.933	0.934	0.936	0.934	0.934	0.934	0.934
CON_T	0.840	0.886	0.904	0.909	0.915	0.923	0.932	0.935	0.936	0.936	0.942

Table 5. Measurements for Tomcat using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.910	0.917	0.923	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.923
PR	0.811	0.800	0.815	0.834	0.847	0.852	0.853	0.858	0.858	0.858	0.927
PR_U	0.910	0.921	0.931	0.933	0.933	0.932	0.933	0.932	0.932	0.932	0.932
CON_T_W	0.914	0.920	0.924	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926
CON_T	0.868	0.906	0.930	0.936	0.937	0.938	0.938	0.938	0.938	0.938	0.939

Table 6. Measurements for Hibernate using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.954	0.957	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958
PR	0.929	0.929	0.933	0.939	0.939	0.946	0.947	0.947	0.947	0.947	0.949
PR_U	0.942	0.947	0.948	0.949	0.949	0.950	0.950	0.950	0.950	0.950	0.951
CON_T_W	0.939	0.942	0.943	0.944	0.944	0.945	0.945	0.945	0.945	0.945	0.944
CON_T	0.924	0.933	0.938	0.941	0.941	0.944	0.945	0.945	0.945	0.945	0.946

6.5. Measurements using only logical dependencies

In the previous section, we added in the graph based on which the ranking algorithm works the logical and structural dependencies. In the current section, we will add only the logical dependencies to the graph.

In tables 7, 8, and 9, are presented the results obtained by using only logical dependencies to detect key classes. The measurements obtained are not as good as using logical and structural dependencies combined or using only structural dependencies. But, all the values obtained are above 0.5, which means that a good part of the key classes is detected by only using logical dependencies. As mentioned in section 3, a classifier is good if it has the ROC-AUC value as close to 1 as possible.

One possible explanation for the less performing results is that the key classes may have a better design than the rest of the classes, which means that are less prone to change. If the key classes are less prone to change, this implies that the number of dependencies extracted from the versioning system can be less than for other classes.

Table 7. Measurements for Ant using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.929
PR	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.855
PR_U	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.933
CON_T_W	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.934
CON_T	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.942

Table 8. Measurements for Tomcat using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Previous
PR_U2_W	0.672	0.656	0.645	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.923
PR	0.685	0.643	0.642	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.927
PR_U	0.685	0.643	0.644	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.932
CON_T_W	0.694	0.636	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.926
CON_T	0.654	0.611	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.939

Table 9. Measurements for Hibernate using only logical dependencies

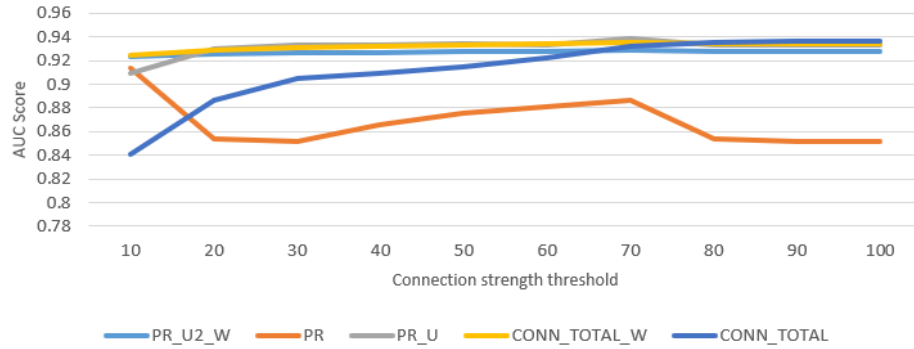
Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.657	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.958
PR	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.949
PR_U	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.951
CON_T_W	0.649	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.944
CON_T	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.946

7. Correlation between details of the systems and results

In this section, we discuss about the correlation between the details of the systems and the results obtained in section 6.

The reason why we are doing this correlation is to find if there are some links between the details of the systems and the results obtained.

The results obtained are presented in figures 5 - 10. We are using plots to display the results obtained to have a clearer view of how the results fluctuate over different thresholds values.

**Fig. 5.** Variation of AUC score when varying connection strength threshold for Ant. Results for structural and logical dependencies combined.

The details of the systems are presented in two tables. In table 10 are the overlappings between structural and logical dependencies expressed in percentages. Each column represents the percentage of logical dependencies that are also structural, for each column the logical dependencies are obtained by applying a different connection strength filter. The connection strength filter begins at 10, meaning that in at least 10 % of the total commits involving two entities, the entities update together. We increase the connection strength filter by 10 up until we reach 100, meaning that in all the commits that involve one entity, the other entity is present also.

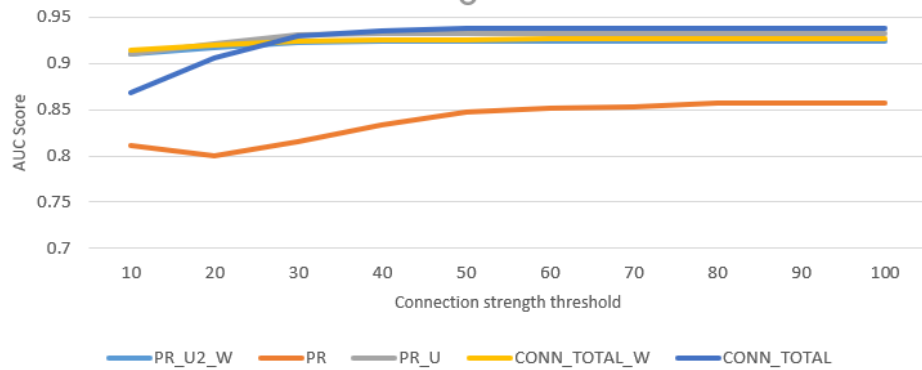


Fig. 6. Variation of AUC score when varying connection strength threshold for Tomcat. Results for structural and logical dependencies combined.

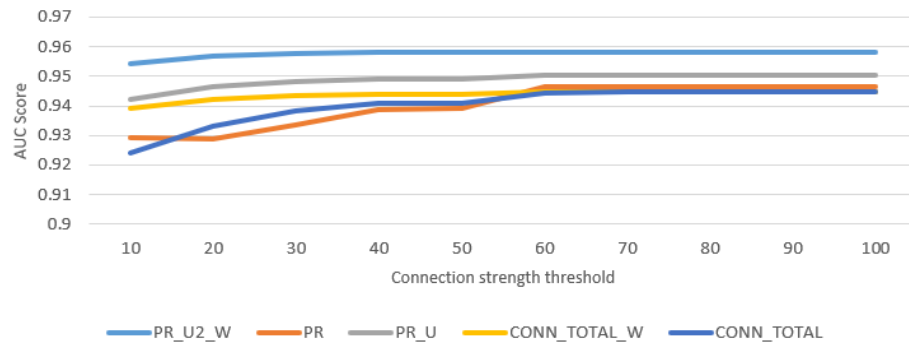


Fig. 7. Variation of AUC score when varying connection strength threshold for Hibernate. Results for structural and logical dependencies combined.

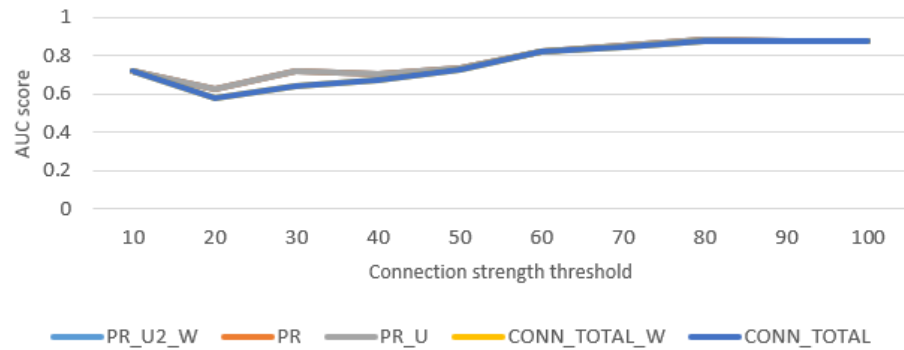


Fig. 8. Variation of AUC score when varying connection strength threshold for Ant. Results for logical dependencies only.

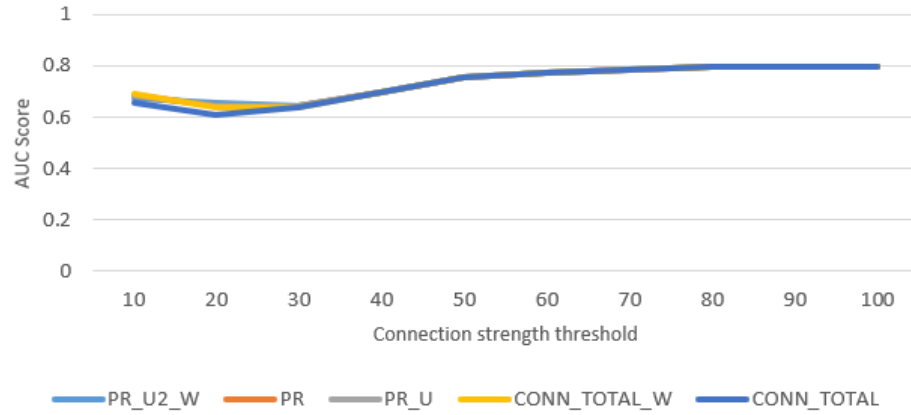


Fig. 9. Variation of AUC score when varying connection strength threshold for Tomcat. Results for logical dependencies only.

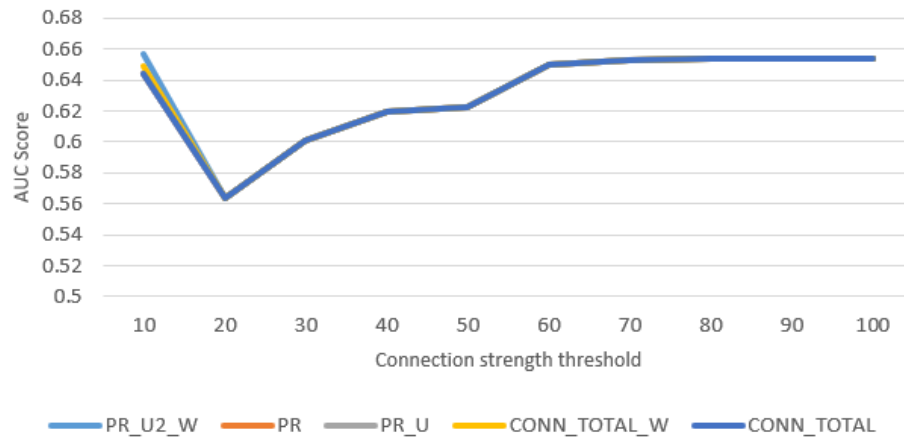


Fig. 10. Variation of AUC score when varying connection strength threshold for Hibernate. Results for logical dependencies only.

1 In table 11 are the ratio numbers between structural dependencies and logical depen-
 2 dependencies. We added this table in order to highlight how different the total number of both
 3 dependencies is.

Table 10. Percentage of logical dependencies that are also structural dependencies

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	25.202	34.419	36.385	34.656	33.528	33.333	28.659	33.333	35.294	35.294
Tomcat Catalina	4.059	22.089	25.000	25.758	25.926	37.525	47.368	55.285	75.000	76.923
Hibernate	6.546	26.607	29.565	32.374	32.543	45.170	44.980	42.473	42.473	42.473

Table 11. Ratio between structural and logical dependencies (SD/LD)

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	1.315	3.284	4.972	5.603	6.175	10.697	12.915	27.154	41.529	41.529
Tomcat Catalina	0.120	0.923	1.313	1.531	1.619	3.177	7.092	13.146	67.375	124.385
Hibernate	1.037	6.391	10.037	14.947	18.940	54.248	83.442	111.704	111.704	111.704

4 In figures 5, 6 and 7 are the measurements obtained by using structural and logical de-
 5 pendencies combined. In all three figures, the measurements at the beginning are smaller
 6 than the rest. Once with the increasing of the threshold value also the measurements begin
 7 to increase. Meaning that better results for key class detection are found. The best mea-
 8 surements are when the threshold value is between 40 and 60, after that, the measurements
 9 tend to decrease a little bit and stay at that fixed value.

10 A possible explanation of the results fluctuation and then capping is that if we are
 11 looking at table 11 we can see that at the beginning, the total number of logical depen-
 12 dencies used is close to the number of existing structural dependencies. The high volume
 13 of logical dependencies introduced might cause an erroneous detection of the key classes,
 14 in consequence, smaller measurements. When the threshold begins to be more restrictive
 15 and the total number of logical dependencies used begins to decrease, the key classes de-
 16 tection starts to improve. This improvement stops after the threshold value reaches 60%.
 17 If we look again at table 11 we can see that after 60% the number of structural depen-
 18 dencies outnumbers the number of logical dependencies up to 124 times in some cases.
 19 In addition, if we look at table 10 we can see that the remaining logical dependencies
 20 overlap a lot with the structural dependencies, so we are not introducing too much new
 21 information.

22 So, the number of logical dependencies used is so small that it doesn't influence the
 23 key class identification. Since the structural dependencies used don't change, we obtain
 24 the same results for different threshold values.

25 In figures 8, 9 and 10 are the measurements obtained by using only logical dependen-
 26 cies. Initially, we expected to see a Gaussian curve, but instead, we see a bell curve. We
 27 think that in the beginning, we use a high number of logical dependencies in key class
 28 detection, among those logical dependencies is an important number of key classes and
 29 also an important number of other classes. But the number of other classes does not influ-
 30 ence the key classes detection. When we start to increase the value of the threshold and

filter more the logical dependencies, we also filter some of the initial detected key classes and remain with a significant number of other classes. In this case, the other classes that remain influence the measurements, causing the worst-performing solutions. Some of the key classes are strongly connected in the versioning system, and even for higher threshold values don't get filtered out. Meanwhile, the rest of the classes that are not key classes get filtered out for higher threshold values which leads to better performing measurements when the threshold value are above 60%.

8. Conclusions

The logical dependencies are filtered co-changing pairs extracted from the versioning system history. The filters applied to the co-changing pairs are the following: the filter based on commit size and the filter based on connection strength.

In our experiments the filter based on commit size had a hard threshold of 10 files, meaning that we use co-changing pairs only from commits with at most 10 files changed. The filter based on connection strength had a variable threshold, starting with 10% and ending with 100%. We used a variable threshold for connection strength because we wanted to observe how this threshold will impact the key classes detection.

In section 6 we approached two scenarios to detect key classes by using logical dependencies. In the first scenario, we used logical dependencies together with structural dependencies and in the second, we used only logical dependencies to detect the key classes. We modified the tool used in the baseline approach to use also logical dependencies, and then we performed the key class identification using that tool. The quality of the results obtained was evaluated with the same tool, the metric used to evaluate the results is Area Under the Receiver Operating Characteristic Curve (ROC-AUC). We then compared the evaluation results with the results obtained by the baseline approach.

Based on the results obtained, compared with the baseline results, we did saw a slight improvement in key class detection when both logical and structural dependencies were used together, the best results were obtained with a connection strength threshold of 40-70%. When we used only logical dependencies to detect key classes, the results were less performing than using only structural or structural and logical dependencies combined.

As we mentioned in section 2, also other researchers tried to identify the key classes, and even though the approaches are not the same, most of them have used the ROC-AUC metric to evaluate the quality of the results. Osman et al. obtained in their research an average Area Under the Receiver Operating Characteristic Curve (ROC-AUC) score of 0.750 [11]. Thung et al. obtained an average ROC-AUC score of 0.825 [18] and Sora et al. (the baseline approach) obtained an average ROC-AUC score of 0.894 [10].

In the current research, we obtained an average ROC-AUC score of 0.926 when using logical and structural dependencies combined and a score of 0.747 when using only logical dependencies to detect key classes.

In conclusion, by using both dependencies combined, we can obtain a slightly better ROC-AUC score than the one obtained by the baseline approach. And, by using only logical dependencies we don't obtain a better score than the baseline approach but compared with the results obtained by other researchers [11], the score obtained is almost equal. The advantage of using only logical dependencies in key class detection is that it

only uses data extracted from the versioning system and can be generalized to various programming languages.

References

1. Bradley, A.P.: The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30(7), 1145–1159 (1997), <https://www.sciencedirect.com/science/article/pii/S0031320396001422>
2. Şora, I.: Helping program comprehension of large software systems by identifying their most important classes. In: *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*. pp. 122–140. Springer International Publishing (2015)
3. Şora, I.: Helping program comprehension of large software systems by identifying their most important classes. In: Maciaszek, L.A., Filipe, J. (eds.) *Evaluation of Novel Approaches to Software Engineering*. pp. 122–140. Springer International Publishing, Cham (2016)
4. Ding, Y., Li, B., He, P.: An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering* 2016, 1–9 (2016)
5. do Nascimento Vale, L., de A. Maia, M.: Keecl: Mining key architecturally relevant classes using dynamic analysis. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. pp. 566–570 (2015)
6. Fawcett, T.: An introduction to roc analysis. *Pattern Recognition Letters* 27(8), 861–874 (2006), <https://www.sciencedirect.com/science/article/pii/S016786550500303X>, rOC Analysis in Pattern Recognition
7. Kamran, M., Ali, M., Akbar, B.: Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics* 10, 21–25 (2016)
8. Meyer, P., Siy, H., Bhowmick, S.: Identifying important classes of large software systems through k-core decomposition. *Adv. Complex Syst.* 17 (2014)
9. Şora, I.: Finding the right needles in hay - helping program comprehension of large software systems. In: *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*, pp. 129–140. INSTICC, SciTePress (2015)
10. Şora, I., Chirila, C.B.: Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology* 116, 106176 (2019), <https://www.sciencedirect.com/science/article/pii/S0950584919301727>
11. Osman, M.H., Chaudron, M.R.V., v. d. Putten, P.: An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: *2013 IEEE International Conference on Software Maintenance*. pp. 140–149 (2013)
12. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999), <http://ilpubs.stanford.edu:8090/422/>, previous number = SIDL-WP-1999-0120
13. Pan, W., Song, B., Li, K., Zhang, K.: Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Generation Computer Systems* 81, 188–202 (2018), <https://www.sciencedirect.com/science/article/pii/S0167739X17302492>
14. Şora, I.: A PageRank based recommender system for identifying key classes in software systems. In: *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics (SACI)*. pp. 495–500 (May 2015)
15. Stana, A.D., Şora, I.: Identifying logical dependencies from co-changing classes. In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*, pp. 486–493. INSTICC, SciTePress (2019)

- 1 16. Steidl, D., Hummel, B., Juergens, E.: Using network analysis for recommendation of cen-
2 tral software classes. In: 2012 19th Working Conference on Reverse Engineering. pp. 93–102
3 (2012)
- 4 17. Tahvildari, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations:
5 a metric-based approach. *J. Softw. Maintenance Res. Pract.* 16, 331–361 (2004)
- 6 18. Thung, F., Lo, D., Osman, M.H., Chaudron, M.R.V.: Condensing class diagrams by analyz-
7 ing design and network metrics using optimistic classification. In: Proceedings of the 22nd
8 International Conference on Program Comprehension. p. 110–121. ICPC 2014, Association
9 for Computing Machinery, New York, NY, USA (2014), [https://doi.org/10.1145/](https://doi.org/10.1145/2597008.2597157)
10 [2597008.2597157](https://doi.org/10.1145/2597008.2597157)
- 11 19. Yang, X., Lo, D., Xia, X., Sun, J.: Condensing class diagrams with minimal manual labeling
12 cost. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMP-
13 SAC). vol. 1, pp. 22–31 (2016)
- 14 20. Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to exe-
15 cution traces to support the program comprehension process. In: Ninth European Conference
16 on Software Maintenance and Reengineering. pp. 134–142 (2005)
- 17 21. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using
18 webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*
19 20(6), 387–417 (2008)