Methods and Tools for the Analysis of Legacy Software Systems

Report 2. Logical dependencies in practice.

PhD Student: Adelina Diana Stana

Department: Calculatoare şi tehnologia informaţiei
PhD Supervisor: Vladimir I. CREŢU

Contents

Chapter 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.

This second report presents results obtained by using the extracted logical dependencies.

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

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

The report is structured as follows: in section 2.1.1 we present the current state of research in identifying key classes. The dataset used in the baseline research and the software systems from that dataset that we have been able to work with is presented in section 2.1.4. The results obtained by the baseline research are presented in section 2.1.3, section 2.1.5 presents the new results obtained by adding logical dependencies. Finally, in section 2.2 the comparison between logical dependencies number and other software metrics is discussed.

Chapter 2

Usage of the extracted logical dependencies

2.1 Identifying key classes using logical dependencies

2.1.1 Definition and state of the art

Zaidman et al [26] were the first to introduce the concept of key classes and it refers to classes that can be found in documents written to provide an architectural overview of the system or an introduction to the system structure. Tahvildari and Kontogiannis have a more detailed definition regarding key classes concept: "Usually, the most important concepts of a system are implemented by very few key classes which can be characterized by the specific properties. These classes, which we refer to as key classes, manage many other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality" [22]. Also, other researchers use a similar concept as the one defined by Zaidman but under different terms like important classes [11] or central software classes [21].

In previous works, the approach for finding key classes is based on ranking the classes with a page ranking algorithm [6], [13], [14], [19] . The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [16]. The PageRank algorithm works based on a recommendation system. If one node has a connection with another node, then it recommends the second node. In previous works, connections are established based on structural dependencies extracted from static code analysis. If A has a structural dependency with B, then A recommends B, and also B recommends A.

2.1.2 Metrics for results evaluation

A classification model is a mapping between expected results and predicted results [9], [3]. Both results can be labeled as positive or negative, which leads us to the confusion matrix from figure 2-1. The confusion matrix has the following outcomes:

Predicted Result / Expected Result	Positive	Negative
Positive	*True Positive*	*False Positive*
Negative	*False Negative*	*True Negative*

Figure 2-1: Confusion matrix

- *true positive*, if the expected result is positive and the predicted result is also positive.
- *false positive*, if the expected result is positive but the predicted result is negative.
- *false negative*, if the expected result is negative but the predicted result is positive.

- *true negative*, if the expected result is negative and the predicted result is also negative.

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:

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

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:

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

To calculate the performance of a classification model, the Receiver Operating Characteristic (ROC) graph can be used. 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).

In multiple related works, the ROC-AUC metric has been used to evaluate the results for finding key classes of software systems [15], [14], [23], [24]. 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.

2.1.3 Baseline approach

In the research of I. Sora et al [14] is used a tool that takes as an input the source code of the system and applies a ranking strategy to rank the classes according to their importance. To differentiate the important classes from the rest of the classes, a TOP threshold for the top classes found is set. The threshold can vary between 20 and 30 classes.

The expected results from the research are based on classes labeled as important classes in the system documentation. The true positives (TP) are the classes found in the reference solution and also in the top TOP ranked classes. False positives (FP) are the classes that are not in the reference solution but are in the TOP ranked classes. True Negatives (TN) are classes that are found neither in the reference solution nor in the TOP ranked classes. False Negatives (FN) are classes that are found in the reference solution but not found in the TOP ranked classes.

In table 2.1 are presented the ROC-AUC values for different attributes computed for the systems Ant, Tomcat Catalina, and Hibernate. We intend to compare these values with the new values obtained by using also logical dependencies in key class detection.

Table 2.1: 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

Class attributes that characterize key classes

In order to identify the key classes of an object-oriented system, we have to determine what metrics can be used in order to get a good overview of the system and its most important classes [7], [26], [17] . The metrics used in previous research can be grouped into the following categories:

- class size metrics: number of fields (NoF), number of methods (NoM), global size (Size = NoF+NoM).
- class connection metrics, any structural dependency between two classes:
  - CONN-IN, the number of distinct classes that use a class;
  - CONN-OUT, the total number of distinct classes that are used by a class;
  - CONN-TOTAL, the total number of distinct classes that a class uses or are used by a class (CONN-IN + CONN-OUT).

- CONN-IN-W, the total weight of distinct classes that use a class.
- CONN-OUT-W, the total weight of distinct classes that are used by a class.
- CONN-TOTAL-W, the total weight of all connections of the class (CONN-IN-W + CONN-OUT-W) [14].

- class pagerank values, previous research use pagerank values computed on both directed and undirected, weighted and unweighted graphs:
  - PR - value computed on the directed and unweighted graph;
  - PR-W - value computed on the directed and weighted graph;
  - PR-U - value computed on the undirected and unweighted graph;
  - PR-U-W - value computed on the undirected and weighted graph;
  - PR-U2-W - value computed on the weighted graph with back-recommendations [6], [13], [14], [19].

Because the extracted logical dependencies from the systems are undirected, from the mentioned metrics, we can use the following ones: CONN-TOTAL, CONN-TOTAL-W, PR-U, PR-U-W and PR-U2-W.

2.1.4 Data set used

To extract the key classes based on logical dependencies, we took the same set of data used in another research involving key class detection. The research of I. Sora et al [14] takes into consideration structural public dependencies that are extracted using static analysis techniques and was performed on the object-oriented systems presented in table 2.2.

The requirements for a system to qualify as suited for investigations using logical dependencies are: has to be on GitHub, has to have release tags to identify the version, and also has to have an increased number of commits. From the total of 14 object-oriented systems listed in the paper [14], 13 of them have repositories in

Github 2.3. And from the found repositories we identified only 6 repositories that have the same release tag as the specified version from table 2.2. It is important to identify the correct release tag for each repository to limit the commits further analyzed by date. Only commits that were made until the specified release are considered and analyzed. The commits number found on the remaining 6 repositories varies from 19108 commits for Tomcat Catalina to 149 commits for JHotDraw. In order to have more accurate results, we need a significant number of commits, so we reached the conclusion that only 3 systems can be used for key classes detection using logical dependencies: Apache Ant, Hibernate, and Tomcat Catalina. From all the systems mentioned in table 2.2 Apache Ant is the most used and analyzed in other works [20], [8], [25], [10].

Table 2.2: Analyzed software systems in previous research paper.

ID	System	Description	Version
Sl	Apache Ant	Java library and command line tool that drive the build processes as targets and extension points depending upon each other	1.6.1
S2	Argo UML	UML modelling tool with support for all UML diagrams.	0.9.5
S3	GWT Portlets	Open source web framework for building GWT (Google Web Toolkit) Applications.	0.9.5 beta
S4	Hibernate	Persistence framework for Java.	5.2.12
S5	javaclient	Java distributed application for playing with robots	2.0.0
S6	jEdit	Java mature text editor for programmers.	5.1.0
S7	JGAP	Genetic Algorithms and Genetic Programming Java library.	3.6.3
S8	JHotDraw	JHotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Java.	6.0b.1
S9	JMeter	JMeter is a Java application designed to load test functional behavior and measure performance	2.0.1
S10	Log4j	Logging Service	2.10.0
S11	Mars	The Mars Simulation Project is a Java project that models and simulates human settlements on Mars planet	3.06.0
S12	Maze	The Maze-solver project simulates an artificial intelligence algorithm on a maze	1.0.0
S13	Neuroph	Neuroph is a Java neural network framework.	2.2.0
S14	Tomcat Catalina	The Apache Tomcat project is an open-source implementation of JavaServlet and JavaServerPages technologies	9.0.4
S15	Wro4J	The Wro4J is a web resource (JS and CSS) optimizer for Java.	1.6.3

Table 2.3: Found systems and versions of the systems in GitHub.

ID	System	Version	Release Tag name	Commits number
Sl	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

2.1.5 Measurements using logical dependencies

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

To determine the connection strength of a pair, we first need to calculate the connection factors for both entities that form a co-changing pair. Assuming that we have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B that contain also entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B}$$

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

After the filtering part, the remaining co-changing pairs, now called logical dependencies, are exported in CSV files.

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

The next step is to use the exported logical dependencies for key classes detection. In order to do that we used the same key class detection tool used in the previous research presented in section 2.1.3. We adapted the tool to be able to process also logical dependencies because previously the tool used only structural dependencies extracted from the source code of the software systems. The workflow is presented in figure 2-2

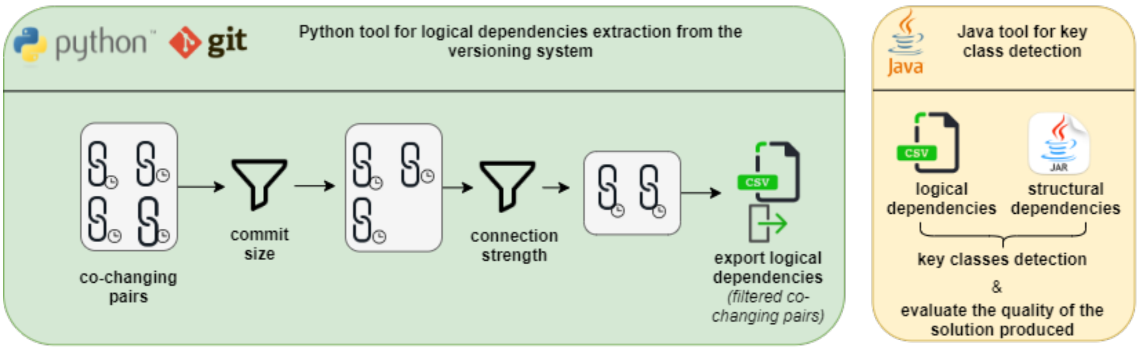

Figure 2-2: Workflow for key classes detection

As we mentioned in the beginning whole purpose is to check if the logical dependencies can improve key class detection. Previously the tool that detects key classes worked only with structural dependencies and now it can also work with logical dependencies. When we introduced the logical dependencies we thought of two use case scenarios for them:

- use logical dependencies together with structural dependencies

- use only logical dependencies

Based on the two scenarios mentioned above we performed key classes detection. Additionally, the tool evaluates the solution produced, the goal is to have ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) metric value as close to 1 as possible.

In tables 2.4, 2.5, and 2.6, we used the logical dependencies together with structural dependencies. On each line, we have the metric that is calculated and on each column, we have the connection strength threshold that was applied to the logical dependencies used in identifying the key classes. We started with logical dependencies that have a connection strength greater than 10%, which means that in at least 10% of the commits involving A or B, A and B update together. Then we increased the threshold value by 10 until we remained only with entities that update in all the commits together. The last column contains the results obtained previously by the tool by only using structural dependencies.

As for the new results obtained by combining structural and logical dependencies, highlighted with orange are the values that are close to the previously registered values but did not surpass them. Highlighted with green are values that are better than the previously registered values. At this step, we can also observe that for all three systems measured in tables 2.4, 2.5, and 2.6, the best values obtained are for connection strength between 40-70%.

Table 2.4: 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

In tables 2.7, 2.8, and 2.9, we only used logical dependencies to detect key classes. The measurements obtained by using only logical dependencies are not as

Table 2.5: Measurements for Tomcat using structural and logical dependencies combined

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 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 2.6: Measurements for Hibernate using structural and logical dependencies combined

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 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

good as using logical and structural dependencies combined or using only structural dependencies. As mentioned in section 2.1.2, 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 2.7: Measurements for Ant using only logical dependencies

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 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 2.8: Measurements for Tomcat using only logical dependencies

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 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 2.9: Measurements for Hibernate using only logical dependencies

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 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

2.1.6 Corelation between details of the systems and results

In this section, we discuss about overlappings between structural and logical dependencies. The structural dependencies are relationships between two elements and indicate that an element of the relationship, in some manner, depends on the other element of the relationship [2], [5].

Structural dependencies can be found by analyzing the source code of the system [18], [4], [1].

Logical dependencies are also relationships between two elements of a software system, but in contrast to structural dependencies, the logical dependencies can be found by analyzing the versioning system history.

It can be the case that a structural dependency is also a logical dependency. This means that the relationship between two elements can be detected via code analysis but also via versioning system analysis.

The reason why we are studying the overlappings between logical and structural dependencies is because we want to see how connected are these two types of dependencies.

In table 2.10 are the overlappings between structural and logical dependencies expressed in percentages for each system mentioned in section 2.1.5. 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.

We can observe that for more restrictive connection strengths we have a higher percentage of logical and structural dependencies overlapping. Especially for Tomcat Catalina, we have an overlapping of 75 % between both types of dependencies for a connection strength of 90%.

In table 2.11 are the ratio numbers between structural dependencies and logical dependencies. We added this table in order to highlight how different the total number of both dependencies is. In this table, we can observe that for more restrictive connection strengths the number of structural dependencies outnumbers the number of logical dependencies by up to 124 times.

Table 2.10: 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 2.11: 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

2.2 Comparison of the extracted data with fan-in and fan-out metric

Fan-in and fan-out are coupling metrics. The fan-in of entity A is the total number of entities that call functions of A. The fan-out of A is the total number of entities called by A [12].

In tables 2.12, 2.13, and 2.14 we can find the metrics detalis for each documented key class of each system. The first column represents the name of each key class, the second column represents the fan_in values for each key class, the third column represents the fan_out values, the fourth column represents the number of entities that call functions of that key class plus the number of entities that are called by the key class (fan_in and fan_out combined), and the fifth column represents the number of logical dependencies in which an entity is involved.

For Ant, we can see in table 2.12 that all the key classes have logical dependencies with other classes. The LD_NUMBER means the number of logical dependencies of an entity. The key classes with the most LD number are Project and IntrospectionHelper, these two entities can be found also in table 2.15 in which we did a top 10 entities that have a logical dependency with other entities. This means that some key classes are involved in software change quite often and can be observed via system history.

Table 2.12: Measurements for Ant key classes

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Target	28	6	34	78
3	UnknownElement	17	13	30	90
4	RuntimeConfigurable	17	13	30	118
5	IntrospectionHelper	18	24	42	143
6	Main	1	13	14	82
7	TaskContainer	11	1	12	21
8	ProjectHelper2$ElementHandler	1	12	13	30
9	Task	110	7	117	88
10	ProjectHelper	16	8	24	101

For Tomcat Catalina, same as for Ant, we can see in table 2.13 that all the key classes have logical dependencies. The key classes with the most LD number are StandardContext and Request, these two entities can also be found in table 2.16 in which we did a top 10 entities that have the most logical dependencies with other entities for Tomcat Catalina.

For Hibernate things are a little bit different, as we can see in table 2.14, key classes like Criterion, Projection, or Transaction have 0 logical dependencies, meaning that those key classes are not involved in any software change. One possible explanation for this is that for Hibernate the architecture is designed in such way that the core is not often touched by change.

In tables 2.15, 2.16, and 2.17 we can find the top 10 entities with logical dependencies. The first column represents the name of each top 10 entity, the second column represents the fan_in values, the third column represents the fan_out values,

Table 2.13: Measurements for Tomcat Catalina key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Context	74	8	82	126
2	Request	48	28	76	215
3	Container	51	8	59	64
4	Response	38	12	50	90
5	StandardContext	11	38	49	216
6	FANector	23	9	32	89
7	Session	29	2	31	28
8	Valve	29	2	31	19
9	Wrapper	29	1	30	36
10	Manager	25	3	28	31
11	Host	26	1	27	44
12	Service	20	6	26	51
13	Engine	23	2	25	1
14	Realm	18	6	24	21
15	CoyoteAdapter	1	22	23	140
16	StandardHost	8	15	23	88
17	LifecycleListener	21	1	22	3
18	StandardEngine	2	19	21	57
19	Pipeline	19	2	21	20
20	Server	16	4	20	49
21	HostConfig	3	15	18	79
22	StandardWrapper	5	13	18	92
23	StandardService	3	12	15	81
24	Catalina	2	13	15	94
25	Loader	14	1	15	18
26	StandardServer	2	12	14	94
27	StandardPipeline	1	10	11	62
28	Bootstrap	3	3	6	41

Table 2.14: Measurements for Hibernate key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	SessionFactoryImplementor	438	43	481	51
2	Type	444	5	449	0
3	Table	89	29	118	82
4	SessionImplementor	52	12	64	14
5	Criteria	45	12	57	15
6	Column	46	10	56	20
7	Session	31	21	52	52
8	Query	12	28	40	0
9	Configuration	1	38	39	115
10	SessionFactory	24	12	36	33
11	Criterion	30	3	33	0
12	Projection	11	3	14	0
13	FANectionProvider	12	2	14	0
14	Transaction	11	1	12	0

the fourth column represents the fan_in and fan_out combined, and the fifth column represents the number of logical dependencies in which the entity is involved.

We did these top 10 tables to offer an overview of the highest registered numbers for LD for each system. As we mentioned before, some of the key classes are also present in these tables, but not all of them.

In table 2.17 we can find the top 10 measurements for Hibernate, most of the table is occupied by inner classes of AbstractEntityPersister. This is expected behavior since class AbstractEntityPersister is also present. This behavior is caused by the impossibility to separate the updates done for a class from its inner classes in the versioning system. So, each time AbstractEntityPersister records a change, also the inner classes are considered to have changed.

Table 2.15: Top 10 measurements for Ant.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Project$AntRefTable	1	2	3	157
3	Path	39	13	52	147
4	Path$PathElement	3	2	5	147
5	IntrospectionHelper	18	24	42	143
6	IntrospectionHelper$AttributeSetter	8	1	9	143
7	IntrospectionHelper$Creator	3	5	8	143
8	IntrospectionHelper$NestedCreator	7	1	8	143
9	Ant	2	15	17	136
10	Ant$Reference	3	1	4	136

Table 2.16: Top 10 measurements for Tomcat Catalina.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	StandardContext	11	38	49	216
2	StandardContext$ContextFilterMaps	0	0	0	216
3	StandardContext$NoPluggabilityServletContext	0	0	0	216
4	Request	48	28	76	215
5	Request$SpecialAttributeAdapter	0	0	0	215
6	ApplicationContext	3	22	25	158
7	ApplicationContext$DispatchData	0	0	0	158
8	ContextConfig	3	26	29	143
9	ContextConfig$DefaultWebXmlCacheEntry	0	0	0	143
10	ContextConfig$JavaClassCacheEntry	0	0	0	143

Table 2.17: Top 10 measurements for Hibernate.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NR
1	AvailableSettings	1	0	1	205
2	AbstractEntityPersister	9	143	152	190
3	AbstractEntityPersister$CacheEntryHelper	0	0	0	190
4	AbstractEntityPersister$InclusionChecker	0	0	0	190
5	AbstractEntityPersister$NoopCacheEntryHelper	0	0	0	190
6	AbstractEntityPersister$ReferenceCacheEntryHelper	0	0	0	190
7	AbstractEntityPersister$StandardCacheEntryHelper	0	0	0	190
8	AbstractEntityPersister$StructuredCacheEntryHelper	0	0	0	190
9	Dialect	265	104	369	176
10	SessionFactoryImpl$SessionBuilderImpl	1	25	26	167

Chapter 3

Conclusions

The purpose of this second report was to use the extracted logical dependencies from the versioning system history. 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 2.1.5 we used logical dependencies together with structural dependencies in order to detect key classes. But, we also used only logical dependencies to detect key classes.

Based on the results obtained we did saw an improvement in key classes 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 not bad but were less performing than using only structural or structural and logical dependencies combined.

Also, in section 2.2 we compared the number of logical dependencies of an entity with the number of entities that call functions of the entity (fan-in metric) and the number of entities called by the entity (fan-out metric). We did not find any direct connection between fan-in, fan-out, and logical dependencies number but we believe that the logical dependencies number can be used complementary together with fan-in and fan-out metric.

Bibliography

[1] David Binkley. Source code analysis: A road map. pages 104–119, 06 2007.

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[3] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[4] Trosky B. Callo Arias, Pieter van der Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, Oct 2011.

[5] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35:864–878, 2009.

[6] Ioana Şora. 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*, pages 122–140. Springer International Publishing, 2015.

[7] Yi Ding, B. Li, and Peng He. An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering*, 2016:1–9, 2016.

[8] L. do Nascimento Vale and M. de A. Maia. Keecle: Mining key architecturally relevant classes using dynamic analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 566–570, 2015.

[9] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[10] M. Kamran, M. Ali, and B. Akbar. Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics*, 10:21–25, 2016.

[11] P. Meyer, H. Siy, and S. Bhowmick. Identifying important classes of large software systems through k-core decomposition. *Adv. Complex Syst.*, 17, 2014.

[12] A. Mubarak, S. Counsell, and R. M. Hierons. An evolutionary study of fan-in and fan-out metrics in oss. In *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pages 473–482, 2010.

[13] Ioana Şora. 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,*, pages 129–140. INSTICC, SciTePress, 2015.

[14] Ioana Şora and Ciprian-Bogdan Chirila. Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology*, 116:106176, 2019.

[15] M. H. Osman, M. R. V. Chaudron, and P. v. d. Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *2013 IEEE International Conference on Software Maintenance*, pages 140–149, 2013.

[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[17] Weifeng Pan, Beibei Song, Kangshun Li, and Kejun Zhang. Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Generation Computer Systems*, 81:188–202, 2018.

[18] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[19] Ioana Şora. 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)*, pages 495–500, May 2015.

[20] Adelina Diana Stana. and Ioana Şora. 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,*, pages 486–493. INSTICC, SciTePress, 2019.

[21] D. Steidl, B. Hummel, and E. Juergens. Using network analysis for recommendation of central software classes. In *2012 19th Working Conference on Reverse Engineering*, pages 93–102, 2012.

[22] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *J. Softw. Maintenance Res. Pract.*, 16:331–361, 2004.

[23] Ferdian Thung, David Lo, Mohd Hafeez Osman, and Michel R. V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 110–121, New York, NY, USA, 2014. Association for Computing Machinery.

[24] X. Yang, D. Lo, X. Xia, and J. Sun. Condensing class diagrams with minimal manual labeling cost. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 22–31, 2016.

[25] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 134–142, 2005.

[26] Andy Zaidman and Serge Demeyer. 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, 2008.