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

In order to obtain logical dependencies, we have to filter the co-changes extracted from the versioning system. The filtering will increase the confidence that the remaining co-changing pairs are indeed logically coupled. And also will help to reduce the size of the co-changes extracted.

We defined 3 methods for filtering the co-changing pairs into logical dependencies. For each method, we analyzed the extracted information from the history of the systems defined in section **??**, and we drew the conclusions presented below.

Filtering based on the size of commit transactions. For this, we filter out each commit transaction that has more files changed than an established threshold. This type of filtering will significantly reduce the amount of co-changing pairs extracted. Big commit transactions (more than 10 files) are rather related to refactoring of names, spellchecks, or file reformating and not to actual code changes. Less than 10% of the total commits are commits with more than 10 files. So filtering out every commit that has more than 10 files changed will not impact so much the size of the studied information from the versioning system. After filtering we are remaining with 90% of commits from which we can extract co-changing pairs.

Filtering based on the number of occurrences. In this step, we filter out each co-changing pair that does not occur in the versioning system more than an established threshold. This type of filtering is meant to increase the confidence

that the extracted co-changing pairs are logically coupled. We decided not to go any further with this type of filtering. Mainly because in our experiments, we observed that if we try to go higher with the occurrences threshold, we risk filtering all the existing co-changing pairs for some small systems. From this filtering method, we concluded that we need to consider the system size and the occurrence trends for each system in particular, rather than setting a "hard" threshold for all the studied systems.

Filtering based on connection strength. The connection strength is calculated based on the connection factors of both entities that form a co-changing pair. For 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 same applies for B with A. This filter type is meant to establish how important, to one another, the entities that form a co-changing pair are. If we filter out all the co-changing pairs that do not update at least half of the time together (factor A and factor B $\geq 50\%$) we remain with a decent quantity of co-changing pairs. And at this point, we can say that the remaining co-changing pairs are logical dependencies.

Based on all the conclusions described above, further in our research, we will use the filter based on commit transaction size (filter out commits that have more than 10 files changed) and the filter based on connection strength.

Chapter 2

Usage of the extracted dependencies

2.1 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 [10] takes into consideration structural public dependencies that are extracted using static analysis techniques and was performed on the object-oriented systems presented in table 2.1.

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 [10], 13 of them have repositories in Github 2.2. And from the found repositories we identified only 6 repositories that have the same release tag as the specified version from table 2.1. 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 2.1 Apache Ant is the most used and analyzed in other works [15], [4], [20], [6].

Table 2.1: 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 2.2: 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

2.2 Identifying key classes using logical dependencies

2.2.1 Definition and previous work

Zaidman et al [21] 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" [17]. Also, other researchers use a similar concept as the one defined by Zaidman but under different terms like important classes [7] or central software classes [16].

In previous works, the approach for finding key classes is based on ranking the classes with a page ranking algorithm [2], [9], [10], [14] . The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [12]. 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.

2.2.2 Attributes for key classes detection

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 [3], [21], [13] . 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) [10].

- 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 [2], [9], [10], [14].

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.

2.2.3 Metrics for results evaluation

A classification model is a mapping between expected results and predicted results [5], [1]. Both results can be labeled as positive or negative, which leads us to the confusion matrix from figure 2-1.

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

Figure 2-1: Confusion matrix

The confusion matrix has the following outcomes:

- *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 [11], [10], [18], [19].

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.

2.2.4 Results obtained by the baseline approach

In the research of I. Sora et al [10] 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 2.3 are presented the ROC-AUC values for different attributes computed for the systems Ant, Tomcat Catalina, and Hibernate.

Table 2.3: 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

2.2.5 Measurements using logical dependencies

To evaluate the results obtained using logical dependencies, we used the same tool used in section X. Previously, the tool used only structural dependencies extracted

from the source code of the software systems. In this chapter, we intend to add also the logical dependencies from the versioning system to observe if the results could be improved or not.

For this, the logical dependencies used were filtered based on the update percentage of the entities involved. We define a logical dependency as a connection observed via commits in the versioning system between entity A and entity B. The update percentage of entity A with entity B is determined as follows: the percentage from the total commits involving A that contains entity B.

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

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

We calculated the update percentage for each side of the connection (LD) and filtered the connections found based on it. The rule set is that both entities had to have an update percentage with each other greater than the threshold value. In tables 2.4, 2.5, and 2.6, we introduced the logical dependencies among structural dependencies. We started with logical dependencies that have a percentage of update grater then 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.

As for the new results obtained, in tables 2.4, 2.5, and 2.6, 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.

Table 2.4: Measurements for Ant using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Previous
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
CONN_T_W	0.924	0.928	0.931	0.932	0.933	0.934	0.936	0.934	0.934	0.934	0.934
CONN_T	0.840	0.886	0.904	0.909	0.915	0.923	0.932	0.935	0.936	0.936	0.942

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

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%	Previous
PR_U2_W	0.912	0.915	0.922	0.923	0.924	0.924	0.923	0.924	0.924	0.924	0.923
PR	0.808	0.785	0.812	0.839	0.844	0.851	0.853	0.857	0.857	0.857	0.927
PR_U	0.912	0.920	0.931	0.932	0.933	0.933	0.933	0.932	0.932	0.932	0.932
CONN_T_W	0.918	0.921	0.924	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926
CONN_T	0.877	0.913	0.932	0.937	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%	Previous
PR_U2_W	0.955	0.957	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958
PR	0.931	0.930	0.936	0.940	0.940	0.946	0.946	0.946	0.946	0.946	0.949
PR_U	0.942	0.946	0.948	0.949	0.949	0.950	0.950	0.950	0.950	0.950	0.951
CONN_T_W	0.939	0.942	0.943	0.944	0.944	0.944	0.945	0.945	0.945	0.945	0.944
CONN_T	0.925	0.933	0.938	0.940	0.941	0.944	0.944	0.944	0.944	0.944	0.946

In tables 2.7, 2.8, and 2.9, we only used logical dependencies. The measurements obtained by using only logical dependencies are not as good as using logical and structural dependencies combined or using only structural dependencies. As mentioned in section 2.2.3, a classifier is good if it has the ROC-AUC value as close to 1 as possible. Our classifier identifies the key classes of a software system, classes that are also called central classes. The central 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%	Previous
PR_U2_W	0.655	0.611	0.650	0.645	0.729	0.797	0.855	0.882	0.865	0.865	0.929
PR	0.655	0.611	0.650	0.645	0.729	0.797	0.855	0.882	0.865	0.865	0.855
PR_U	0.655	0.611	0.650	0.645	0.729	0.797	0.855	0.882	0.865	0.865	0.933
CONN_T_W	0.646	0.523	0.617	0.657	0.722	0.785	0.845	0.878	0.865	0.865	0.934
CONN_T	0.646	0.523	0.617	0.657	0.722	0.785	0.845	0.878	0.865	0.865	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	0.702	0.627	0.627	0.712	0.741	0.775	0.786	0.796	0.796	0.796	0.923
PR	0.675	0.617	0.627	0.712	0.741	0.775	0.786	0.796	0.796	0.796	0.927
PR_U	0.675	0.618	0.627	0.712	0.741	0.775	0.786	0.796	0.796	0.796	0.932
CONN_T_W	0.676	0.597	0.624	0.712	0.741	0.775	0.786	0.796	0.796	0.796	0.926
CONN_T	0.638	0.585	0.624	0.712	0.741	0.775	0.786	0.796	0.796	0.796	0.939

Table 2.9: Measurements for Hibernate using only logical dependencies

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%	Previous
PR_U2_W	0.651	0.596	0.597	0.616	0.619	0.644	0.649	0.650	0.650	0.650	0.958
PR	0.641	0.594	0.597	0.616	0.619	0.644	0.649	0.650	0.650	0.650	0.949
PR_U	0.641	0.595	0.597	0.616	0.619	0.644	0.649	0.650	0.650	0.650	0.951
CONN_T_W	0.652	0.591	0.597	0.616	0.619	0.644	0.649	0.650	0.650	0.650	0.944
CONN_T	0.649	0.591	0.597	0.616	0.619	0.644	0.649	0.650	0.650	0.650	0.946

2.3 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 modules that call functions of A. The fan-out of A is the total number of entities called by A [8]. Related to fan-in and fan-out we have extracted CONN_IN and CONN_OUT.

In tables 2.10, 2.11, and 2.12 we can find the metrics detalis for each documented key class.

Table 2.10: Measurements for Ant key classes

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
1	Project	191	191	214	157
2	Target	28	28	34	78
3	UnknownElement	17	17	30	90
4	RuntimeConfigurable	11	11	19	118
5	IntrospectionHelper	18	18	42	143
6	Main	1	1	14	82
7	TaskContainer	11	11	12	21
8	ProjectHelper2$ElementHandler	1	1	13	30
9	Task	110	110	117	88
10	ProjectHelper	16	16	24	101

In tables 2.13, 2.14, and 2.15 we can find the top 10 'best ranked' logical dependencies. As we can observe, the entities have only few structural connections.

Table 2.11: Measurements for Tomcat Catalina key classes.

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
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	Connector	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 2.12: Measurements for Hibernate key classes.

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
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	ConnectionProvider	12	2	14	0
14	Transaction	11	1	12	0

Highlighed with orange are the key classes found in top 10. To be continued....

Table 2.13: Top 10 measurements for Ant.

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
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 2.14: Top 10 measurements for Tomcat Catalina.

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
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 2.15: Top 10 measurements for Hibernate.

Nr.	Classname	CONN_IN	CONN_OUT	CONN_TOTAL	LD
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

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.