A PageRank Based Recommender System for Identifying Key Classes in Software Systems

Ioana Şora
Department of Computer and Software Engineering,
Politehnica University of Timisoara, Romania
ioana.sora@cs.upt.ro

***Abstract*—Program comprehension is a fundamental prerequisite before software engineers may engage in software maintenance or evolution activities and requires the study of large amounts of documentation - either developer documentation or reverse engineered. Very often, from this documentation is missing a short overview document pointing to the most important classes of the system, these who are essential for starting the understanding of the systems architecture.**

In this work we propose a recommender tool to automatically identify the most important classes of a system. Our approach relies on modeling the static dependencies structure of the system as a graph and applying a graph ranking algorithm. We empirically identify the optimal way of building the system graph, identifying how different dependency types should be taken into account. In experiments performed on a set of open source real life systems, we compare the sets of classes recommended by our tool with these included in the architectural overviews provided by the system developers.

I. Introduction

Program comprehension [1] is the first step in any activity of software maintenance or evolution. It is supported by documentation, either developer documentation or reverse engineered documentation. Reverse engineering a large software system often produces a huge amount of information, whose comprehension or further processing would take a long time. A class diagram, reverse engineered from a system with hundreds or even thousands of classes is of little use when trying to understand the system in absence of any other documentation. Even when documentation is available, it may be too detailed and scattered, such as the one generated by *javadoc* from all the classes and packages of the system.

These kinds of documentation mentioned above do not provide the information structured in such a way that is easying the initial effort of the maintainer working with a system which is new and unknown to him. In order to manage the complexity of all the details one would need a short document giving an architectural overview. Architectural overviews may be given either as abstractive summaries, introducing higher-level abstractions, or as extractive summaries, pointing directly to a small number of important classes of the system. An architectural overview given in the form of an extractive summary is the more useful form for getting started with program comprehension as a preparation for maintenance activities.

In this paper we propose a recommender tool for finding the most important classes of a software system. It can be used either as a stand-alone tool or as a plugin in IDE's or software visualization tools, in order to help software engineers in tasks related to incipient program comprehension.

Our approach of identifying the most important classes of a software project is based on ranking them with a graph-ranking algorithm which adapts PageRank [2]. We model the software system as a graph having classes as nodes and use PageRank to rank them. A cut threshold is later used to delimit the top ranked classes which are the recommended summary.

PageRank is a graph-based ranking algorithm well known for its key contribution to the Web search technology, by providing a Web page ranking mechanism. The basic idea of the algorithm is that of *voting* or *recommendation*. When one node links to another one, it is considered that it gives a recommendation (a vote) for that other node. A node is important if it receives votes from many other nodes. Also, the importance of the vote is determined by the importance of the node giving the vote. It results that the importance of a node is determined by both the number of votes that it receives and the importance of the nodes giving these votes. Although the original PageRank definition [2] works on unweighted graphs, there are versions that have been adapted to work on weighted graphs.

The PageRank algorithm and different versions of it have also been used in many other application domains: citation ranking and journal impact factors [3], in various recommender systems such as [4], and in the field of natural language processing for automatic text summarization. [5], [6].

In software engineering, there have been studies that have applied PageRank to software entities: Coderank [7] introduces the idea of computing PageRank values for software artifacts such as classes or functions of a project. Componentrank [8] uses PageRank values for retrieving the most useful software components for reuse from multiple software libraries. Although showing some similarities, this problem is different from that of identifying the key classes of a software system. There is no experimental validation with identifying key components of real software systems. However, for a practical usable tool for program comprehension, it is crucial to find out which is the best way to model the software system as a graph in order to obtain good results for this kind of application. Also it must be experimentally proved on real life case studies that the results produced by the tool are similar with the opinions of human experts.

In Section II we introduce our approach of modeling the structure of object oriented software systems by static dependencies and our hypothesis on using this information for identification of most important classes. Section III presents experimental results of applying our approach to a set of relevant open-source projects and finetuning the model parameters. Section IV discusses our results by comparing with related work. Section V draws the conclusions of this paper.

II. Modeling recommendation relationships between classes

A. Static dependencies between classes

Static dependencies in object oriented languages are produced by various situations. Following categories of dependencies can exist from a class A to a class (or interface) B:

- inheritance: A extends B.
- realization: A implements B.
- field: A has at least one field of type B.
- member access: Code belonging to A accesses a field of B
- method call: A calls a method of B. Every distinct method of B which is called is counted as a new dependency.
- parameter: A method of A has at least one parameter of type B
- return type: A method of A has a return type B
- local variable: A local variable of type B is declared in code belonging to A
- instantiation: An instance of B is created in code belonging to A
- cast: A type-cast to B occurs in code belonging to A

Several different kinds of dependency relationships may be summed up between two classes: for example, A can have a field of type B, instantiate objects of this type, while also have a method with parameters of type B and call several different methods of B.

We identify all these categories of dependencies by static analysis with help of the model extractors of the ART tool suite [9].

The dependency relationship between two classes is characterized by a dependency strength, which takes into account both the number of dependency relationships and the types of dependency relationships between the two classes. We estimate the strength of a dependency using a summative approach based on weights which reflect the ordering of dependency types according to their relative importance. Establishing the weights (the relative importance of static dependency types) is a subject of empirical estimation in the context of a certain goal and different authors use different frameworks for this [10]. In Section III we will empirically determine the optimal values of the weights in the context of identification of key classes.

B. Modeling recommendation relationships between classes

The software system is modeled as a graph having as nodes classes or interfaces. A directed edge from node A to node B means that node A recommends node B as important. Finding where and how to place the recommendation edges is the enabling element for an effective class ranking approach.

In our model, first introduced in [11], the recommendations derive from the static program dependencies. If A depends on B, this means both that A gives a recommendation to B but also that B gives a recommendation to A. We call the edge from A to B a *forward recommendation*, while the edge from B to A is a *back recommendation*.

The forward recommendations, resulting directly from dependencies, are motivated by the fact that a class which is used by many other classes has good chances to be an important one, representing a fundamental data or business model. But also back recommendations are needed to reflect the case when a class which is using a lot of other important classes is an important one, such as a class containing a lot of control of the application or an important front-end class. If only the directed dependency would be considered as a recommendation, then library classes would rank very high while the classes containing the control would rank low.

Recommendations also have weights. A class is not recommending all its dependency classes with an equal number of votes. It will give more recommendation votes to these classes that offer it more services. Thus recommendation weights are derived from the type and amount of dependencies, and can be different for forward recommendations and back recommendations.

The weight of the forward recommendation from A to B is given by the dependency strength of the dependency relationship from A to B. The weight of the back recommendation from B to A is a fraction F of the weight of the forward recommendation from A to B. Section III investigates the optimal value for this fraction, correlated with the optimal values of the dependency weights.

C. Research questions

The research questions to be empirically answered in Section III, in order to have the right model for a good recommender tool, are the following ones:

- Which is the role of dependency directions ?
  - Are back-recommendations needed ?
  - Which is the relative contribution of back-recommendations compared to that of forward recommendations ?
- Which types of dependencies are relevant for the goal of this recommender tool ? We will explore which one of the following hypothesis holds:
  - All dependency types are equally important ?
  - All dependency types are important but in different proportions and these have to be determined ?
  - Some dependency types can be ignored without affecting the quality of the result or even improving it ?
- Which parts of the ranking are relevant for the goal of the recommender tool ?

III. Empirical determination and validation of model parameters

A. Experimental setup and approach

We have chosen as case studies a set of relevant open source systems where design documentation is available such that a set of key classes can be given as a reference solution to be compared with the solutions produced by our recommender tool.

For this kind of tool, recall and precision are defined in [12] as follows: The *recall*, showing the techniques retrieval power, is computed as the percentage of key classes retrieved by the technique versus the total number of key classes present in the reference set. The *precision*, showing the techniques retrieval quality, is computed as the percentage of key classes retrieved versus the total size of the result set.

We run our tool that implements the ranking approach described in section II, using weighted recommendations, according to the type and amount of dependencies as well as back-recommendations.

Our goal is to fine-tune the values for the model parameters (dependency types and weights and dependency directions), such that our tool achieves its best recall and precision. We experiment with following sets of values:

For dependency types and weights, we consider the following variants:

- *AllDep*: all dependency types, with equal contributions
- *CallsOnly*: only method calls are considered, ignoring all other types of dependencies. The number of distinct methods called is taken into account in the global dependency strength.
- *InterfOnly*: only relationships induced by elements visible from the interface are counted (inheritance, implementation, method parameters), ignoring all details such as local variables, member accesses and method calls.
- *AllWeighted*: all dependency types, but weighted such that interface elements have a higher weight than method calls while local variables use brings the smallest weights.

For dependency directions, we consider the following variants:

- F=0: no back recommendation, only forward recommendation determined by the dependency relationship.
- F=1: back-recommendations have the same weight as forward recommendations.
- F=1/2, F=1/4: back-recommendations have a smaller weight than forward recommendations

We study all the combinations of resulting from values for dependency weights and the fraction F of back-recommendations.

B. Case studies

We chose as case studies for this work four open source real life systems: JHotDraw, jEdit, Ant and JMeter. The choice of these case studies was guided by the requirement that design documentation is available, from their developers and if possible also documented in other independent studies.

JHotDraw[1] is a highly customizable graphic framework for structured drawing editors, and we analyze here release 6.0b.1. Its core has 398 classes. We consider that the set of key classes of JHotDraw is formed by these pointed out by the developers, completed with the some classes added in the study of [13]: `Figure`, `Drawing`, `DrawingView`, `DrawApplication`, `Tool`, `Handle`, `DrawingEditor`, `StandardDrawingView`, `CompositeFigure`. This set of 9 classes is further considered the reference solution.

JEdit[2] is a cross platform programmer's text editor written in Java. We analyze the code of release 5.1.0, with 1266 classes. Developer documentation is available[3] and it points out the following classes of interest: `jEdit`, `View`, `EditPane`, `Buffer`, `JEditTextArea`, `Log`, `EBMessage`.

Apache Ant[4] is a Java library and command-line tool to compile, build, test and run Java applications. We analyze release 1.6.1, feeding as input ant.jar containing the core part of ant. It contains 524 classes. Design documentation[5] and other works that analyzed this system [12] mention following important classes: `Project`, `Target`, `UnknownElement`, `RuntimeConfigurable`, `Task`, `IntrospectionHelper`, `ProjectHelper2`, `ProjectHelperImpl`.

Jakarta JMeter[6] is a Java application for testing of Web Applications. We analyze version 2.0.1, its core found in ApacheJMeter_core.jar which contains 280 classes. Design documentation[7] and other works that analyzed this system [12] mentions following classes: `AbstractAction`, `JMeterEngine`, `JMeterTreeModel`, `JMeterThread`, `JMeterGUIComponent`, `Sampler`, `SampleResult`, `TestCompiler`, `TestElement`, `TestListener`, `TestPlan`, `TestPlanGUI`, `ThreadGroup`.

C. Identifying the optimal values for model parameters

We have run our tool on all the four software systems, for combining four sets of dependency weights (*AllDep*, *AllWeights*, *CallsOnly*, *InterfOnly*) and four values for F, the fraction of back-recommendations. The results are summarized in Figure 1.

For every system and every combination of tool parameters, we count the percentage of classes from the reference set that have been retrieved (the recall of the tool) in the first 10 up to the first 50 ranked classes. We establish these threshold independent from the size of the analyzed system: for a large system, even a small percentage threshold such as 10% would corespond to about 100 classes, and such a large recomended set would be of no practical utility in the scope of this work. Also we record the index (ranking) where the last class of the reference set has been found.

[1]http://www.jhotdraw.org/

[2]http://jedit.org/

[3]http://community.jedit.org/cgi-bin/TWiki/view/Main/JEditSourceCodeIntro

[4]http://ant.apache.org/

[5]http://codefeed.com/tutorial/ant_config.html

[6]http://jmeter.apache.org/

[7]http://wiki.apache.org/jmeter/

		JHotDraw						Ant			
		F=0	**F=1**	**F=1/2**	**F=1/4**			**F=0**	**F=1**	**F=1/2**	**F=1/4**
AllDep	Top 10	0.22	0.56	0.56	0.44		Top 10	0.38	0.38	0.25	0.38
	Top 15	0.44	0.67	0.78	0.78		Top 15	0.63	0.63	0.63	0.63
	Top 20	0.56	0.89	0.89	0.89		Top 20	0.63	0.88	1.00	1.00
	Top 30	0.67	0.89	0.89	1.00		Top 30	0.75	1.00	1.00	1.00
	Top 50	0.67	1.00	1.00	1.00		Top 50	0.75	1.00	1.00	1.00
	LastAt	*92*	*43*	*33*	*27*		*LastAt*	*201*	*29*	*19*	*19*
AllWeigh	Top 10	0.22	0.56	0.44	0.44		Top 10	0.38	0.38	0.25	0.38
	Top 15	0.22	0.44	0.78	0.78		Top 15	0.50	0.63	0.63	0.75
	Top 20	0.56	0.89	0.89	0.89		Top 20	0.63	0.88	1.00	1.00
	Top 30	0.67	0.89	0.89	1.00		Top 30	0.75	1.00	1.00	1.00
	Top 50	0.67	1.00	1.00	1.00		Top 50	0.75	1.00	1.00	1.00
	LastAt	*87*	*31*	*25*	*21*		*LastAt*	*256*	*26*	*17*	*20*
CallsOnly	Top 10	0.33	0.56	0.56	0.44		Top 10	0.38	0.38	0.25	0.25
	Top 15	0.44	0.67	0.67	0.67		Top 15	0.44	0.38	0.50	0.50
	Top 20	0.56	0.78	0.89	0.89		Top 20	0.75	0.63	0.88	0.88
	Top 30	0.78	0.89	0.89	0.89		Top 30	0.67	0.89	0.89	1.00
	Top 50	0.89	1.00	1.00	1.00		Top 50	0.75	1.00	1.00	1.00
	LastAt	*64*	*46*	*48*	*50*		*LastAt*	*230*	*48*	*50*	*28*
InterfOnly	Top 10	0.33	0.67	0.67	0.56		Top 10	0.50	0.25	0.25	0.38
	Top 15	0.56	0.78	0.67	0.67		Top 15	0.63	0.25	0.38	0.38
	Top 20	0.67	0.78	0.78	0.89		Top 20	0.75	0.38	0.38	0.50
	Top 30	0.67	1.00	1.00	1.00		Top 30	0.75	0.63	0.63	0.75
	Top 50	0.78	1.00	1.00	1.00		Top 50	0.75	0.75	0.75	0.88
	LastAt	76	27	28	29		LastAt	252	253	248	249

		jEdit						jMeter			
		F=0	**F=1**	**F=1/2**	**F=1/4**			**F=0**	**F=1**	**F=1/2**	**F=1/4**
AllDep	Top 10	0.71	0.86	0.86	0.71		Top 10	0.15	0.31	0.31	0.31
	Top 15	0.86	0.86	0.86	0.86		Top 15	0.31	0.46	0.46	0.38
	Top 20	1.00	0.86	0.86	0.86		Top 20	0.38	0.46	0.54	0.46
	Top 30	1.00	0.86	0.86	0.86		Top 30	0.46	0.54	0.69	0.69
	Top 50	1.00	0.86	0.86	1.00		Top 50	0.62	0.85	0.77	0.77
	LastAt	*19*	*160*	*97*	*50*		*LastAt*	*140*	*84*	*91*	*102*
AllWeigh	Top 10	0.86	0.71	0.71	0.71		Top 10	0.15	0.31	0.31	0.23
	Top 15	0.86	0.86	0.86	0.86		Top 15	0.31	0.38	0.38	0.38
	Top 20	1.00	0.86	0.86	0.86		Top 20	0.38	0.46	0.54	0.54
	Top 30	1.00	0.86	0.86	1.00		Top 30	0.38	0.62	0.62	0.62
	Top 50	1.00	0.86	1.00	1.00		Top 50	0.69	0.85	0.69	0.69
	LastAt	*18*	*93*	*53*	*29*		*LastAt*	*121*	*75*	*88*	*77*
CallsOnly	Top 10	0.57	0.86	0.86	0.71		Top 10	0.23	0.31	0.31	0.31
	Top 15	0.57	0.86	0.86	0.86		Top 15	0.31	0.31	0.31	0.38
	Top 20	0.71	0.86	0.86	0.86		Top 20	0.38	0.31	0.31	0.46
	Top 30	0.71	0.86	0.86	0.86		Top 30	0.46	0.54	0.54	0.54
	Top 50	0.71	0.86	0.86	0.86		Top 50	0.69	0.77	0.69	0.62
	LastAt	*137*	*454*	*427*	*364*		*LastAt*	*121*	*114*	*94*	*94*
InterfOnly	Top 10	0.43	0.57	0.43	0.43		Top 10	0.23	0.31	0.31	0.23
	Top 15	0.43	0.86	0.71	0.71		Top 15	0.23	0.38	0.31	0.31
	Top 20	0.57	0.86	0.71	0.86		Top 20	0.46	0.54	0.38	0.38
	Top 30	0.57	0.86	0.86	0.86		Top 30	0.54	0.62	0.62	0.54
	Top 50	0.86	0.86	0.86	0.86		Top 50	0.85	0.85	0.85	0.85
	LastAt	1264	1263	1263	1263		LastAt	118	95	110	129

Fig. 1. Recall measured when varying the values for the dependency types weights and the fraction of back-recommendations.

Analyzing the results in Figure 1, we can give following answers to the research questions stated in subsection II-C:

- Back-recommendations are certainly needed. With the exception of the system jEdit, less classes from the key set are found when F=0(without back-recommendations). Regarding the optimal value of F, it appears however that it is the smaller values (F=1/4 or F=1/2) that will bring an overall positive result in all cases, better than the case F=1. It is however difficult to decide, on hand of the current set of experiments, which one of the values F=1/4 or F=1/2 is better.
- The dependency types that bring the best results may differ from system to system, depending whether more key classes are fundamental data models or whether they contain a lot of control. This set of experiment shows that the best approach is to take into account all dependency types (*AllDeps* or *AllWeighted*) since it brings overall the best results. More experimental investigation and fine-tuning of the case *AllWeighted*, as we have it intuitively proposed initially in [11], could bring further optimizations.
- From all the experiments, it results that it is reasonable to limit the tool recommended set to the top 20 or top 30 ranked classes, independent from the size of the system. For all systems we noticed that a couple of classes, the most important ones, always appear among the top 10, almost independent from the parameters of the tool. The majority of the other classes from the reference sets are found in the top 20 or top 30, in different orders for different parameters. A very few classes from certain case studies could not be found within the considered cut thresholds, but their individual examination showed that their presence in the top summary is indeed arguable.

D. Tool evaluation

According to the conclusion drawn in the previous subsection, we set the parameters of our tool such that we take into account all dependency types, with different weights (the *AllWeighted* case) and back-recommendations are assigned a fraction F=1/4. The cut threshold is set at the top 20 ranked classes. With these values, we obtain following evaluation of our tool, summarized in table I:

TABLE I
EXPERIMENTAL RESULTS SUMMARY

	JHotDraw	jEdit	Ant	JMeter
System size	398	1266	524	280
Reference size	9	7	8	13
Hits in Top 20	8	6	8	7
Execution time	1 min	3 min	2 min	1 min

From all test cases, we can compute the average values for precision and recall, which are 0.35 respectively 0.82.

We consider that the recall is the relevant metric for the evaluation of the tool. The precision metrics is not so relevant in this case since it considers true positives only classes from a fixed reference set which has an average size of 10 classes. However, when examining the top 20 ranked classes, we can observe that there are among them other classes which are certainly not unimportant. Another approach of validation of the tool results would be to show the output of the tool to a set of experts that know well the systems under analysis and ask them to identify the classes that rightfully belong to the top summary. However, this experiment can be hardly carried out in practice in an objective and accurate way, thus we remained with the unfavorable but more objective way of comparing with a fixed reference solution.

Program comprehension is effectively supported by our tool, which recommends a small number of classes identified as key classes that can be used as starting points for further analysis. For practical effectiveness, it is not worth to lower the cut threshold. Also, a general ranking of all the classes of a system is not possible, since in the median region of the ranking there are classes which are important for certain features of the system but not for the system as a whole. It makes no sense to try to rank these classes with each other in a general ranking, their analysis is better done with techniques such as feature localization [14].

IV. Related Work

There are a few approaches trying to identify the most important classes from a software system. They use as input information extracted either by static analysis [15], [16] or by dynamic analysis [17]. The techniques that have been applied for identifying the key classes are based on webmining techniques [12], network analysis [16], and machine learning [15], [18].

Zaidman et al [12], [19], [17], uses another graph-ranking algorithm known from webmining, HITS, in order to detect key classes of a software system. They apply HITS on models built by dynamic and static analysis, and perform experiments on two case studies. Using dynamic analysis they obtain an average recall of 92% and precision 46%. The major drawback of their method is that dynamic analysis relies very much on the user finding good execution scenarios. It also has scalability issues and a high execution time (1h45). For this reason, they trace only a small number of classes (for Ant only 127 classes were traced, preselected by the user for this role). Zaidman also combined this webmining technique with static analysis but concluded that the static analysis was not able to achieve a reasonable precision and recall. Here their best results were a recall of 50% and precision of 8%, while the execution time is still high (over 1 hour).

In this work we prove that static analysis can be used to successfully and efficiently identify key classes. Our results near the values obtained by [12] with dynamic analysis, while the execution time in our case is only 1-2 minutes. Also, the user is not required to know execution scenarios, he just provides the code to be analyzed without any other intervention, even as jars from the binary distribution. This makes our tool really usable in practice. We consider that our positive result is mainly due to our recommendation model, which takes into account all possible types of static dependencies with appropriate weights, while Zaidman uses coupling metrics that are built from method calls only. We also appreciate in the work of [12] the extensive description of their result sets in case of Ant and JMeter, which allowed us to compare with our result sets for these two systems.

Steidl et al describe another approach that starts from static analysis in [16]. The importance of classes is given by their centrality index computed on the the dependency graph. They found that using the Markov centrality leads to the best results, reporting a precision between 60% and 80%, but only in the top 10 recommendation set, while this precision drops dramatically in the top 50 set. Their experiments were performed on a set of 4 systems. From the data presented, the reference solutions were the reunions of different expert opinion instead of intersection of such. Theses larger baseline solutions may have favored the count of hits in the top 10, as opposed to the smaller and fixed reference solutions used in our experiments. From the results presented, we globally appreciate the retrieval power of this technique is similar with ours.

A different approach, based on machine learning, for identifying the important classes of a system with the goal of condensing reverse engineered class diagrams is presented in [15] and [18]. Available forward design diagrams are used to learn and then to validate the quality of a set of prediction algorithms. In [18], design metrics and network metrics are independently computed and used as attributes by the learning algorithms. In our approach, we combine the pagerank network metric and the design metrics, by computing pagerank on the weighted graph resulting from dependency-based design metrics.

V. Conclusions

In this paper, we propose a method for identifying the key classes of a software system by modeling the system as a graph built by static analysis of program dependencies and applying the PageRank algorithm for ranking its nodes.

The key for the effectiveness of our approach is how the graph is built: it takes into account all types of static dependencies between classes, weighted according to the relative importance given by the dependency type and number of occurrences. Also, it is important to have edges for both forward and backward recommendations. We have empirically determined and validated the values for these parameters.

The experiments done on real systems show good results, proving the practical effectiveness of our tool, which gives the user a good start for program comprehension, providing him easy and quickly with a trustworthy and short recommendation set including the key classes which form the executive summary of the system.

References

[1] A. von Mayrhauser and A. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, Aug 1995.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[3] N. Ma, J. Guan, and Y. Zhao, “Bringing pagerank to the citation analysis,” *Information Processing & Management*, vol. 44, no. 2, pp. 800 – 810, 2008, evaluating Exploratory Search Systems Digital Libraries in the Context of Users Broader Activities.

[4] L. Zhang, K. Zhang, and C. Li, “A topical pagerank based algorithm for recommender systems,” in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’08. New York, NY, USA: ACM, 2008, pp. 713–714.

[5] G. Erkan and D. R. Radev, “Lexrank: Graph-based lexical centrality as salience in text summarization,” *J. Artif. Intell. Res.(JAIR)*, vol. 22, no. 1, pp. 457–479, 2004.

[6] R. Mihalcea and P. Tarau, “Textrank: Bringing order into texts,” in *Proceedings of EMNLP 2004*, D. Lin and D. Wu, Eds. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 404–411.

[7] B. Neate, W. Irwin, and N. Churcher, “Coderank: a new family of software metrics,” in *Software Engineering Conference, 2006. Australian*, April 2006, pp. 10 pp.–378.

[8] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, “Ranking significance of software components based on use relations,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 213–225, March 2005.

[9] I. Şora, “Unified modeling of static relationships between program elements,” in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science, L. Maciaszek and J. Filipe, Eds. Springer Berlin Heidelberg, 2013, vol. 410, pp. 95–109.

[10] L. Briand, J. Daly, and J. Wust, “A unified framework for coupling measurement in object-oriented systems,” *Software Engineering, IEEE Transactions on*, vol. 25, no. 1, pp. 91–121, Jan 1999.

[11] I. Şora, “Finding the right needles in hay - helping program comprehension of large software systems,” in *Proceedings of 10th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2015.

[12] A. Zaidman and S. Demeyer, “Automatic identification of key classes in a software system using webmining techniques,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 387–417, 2008.

[13] Y.-G. Guéhéneuc, “A reverse engineering tool for precise class diagrams,” in *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’04. IBM Press, 2004, pp. 28–41.

[14] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[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 *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 140–149.

[16] D. Steidl, B. Hummel, and E. Juergens, “Using network analysis for recommendation of central software classes,” in *19th Working Conference on Reverse Engineering (WCRE)*, Oct 2012, pp. 93–102.

[17] 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, CSMR 2005.*, March 2005, pp. 134–142.

[18] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron, “Condensing class diagrams by analyzing design and network metrics using optimistic classification,” in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 110–121. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597157

[19] A. Zaidman, B. Du Bois, and S. Demeyer, “How webmining and coupling metrics improve early program comprehension,” in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 74–78.