Using Fuzzy Rules for Identifying Key Classes in Software Systems

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

***Abstract*—In order to help software engineers deal with the increasing size and complexity of software systems, current research is directed toward developing more intelligent tools such as recommendation systems for software engineering.**

In this paper we propose a tool to help software engineers in tasks related to early stages of program comprehension, by finding the most important classes of a software system.

Our approach of identifying the most important classes of a software project is based on ranking them by using fuzzy rules that have in their premises attributes of the classes. These attributes are the size of the class, the weighted incoming and outgoing dependencies, and the PageRank value of the class in the graph created from class dependencies. Experiments done on a set of well-known open source real-life systems produce good results for identifying the key classes of software systems.

I. Introduction

Software engineers have to deal in all phases of the software process with problems induced by the increasing size and complexity of software systems. Tools like integrated development environments (IDEs), modeling tools, collaborative and versioning tools are since a long time indispensable in their work. Current research is directed toward developing more intelligent tools such as recommendation systems for software engineering. These are tools that help software engineers master the large amount of code, models, documentation etc. that they must navigate and that assist them with a wide range of their activities, from reusing code to writing effective bug reports [1].

Program comprehension [2] is the activity of understanding how a software system or a part of it works. It is an important software engineering activity, which is necessary to facilitate reuse, maintenance, reengineering or extension of existing software systems. In the case of large software systems, program comprehension has to deal with the huge amount of code that implements them. When starting with the study of an unknown system, software engineers are overwhelmed by the amount of information, which makes it difcult and time consuming to filter out the important elements from a lot of details.

In this paper we propose a recommendation tool to help software engineers in tasks related to early stages of program comprehension, by finding the most important classes of a software system. It can be used either as a stand-alone tool, or as a plug-in in IDEs where it can point out to the developer the most important classes in legacy source code, or in software reverse engineering and visualization tools for pruning class diagrams by filtering unimportant classes.

Our approach of identifying the most important classes of a software project is based on ranking them by using fuzzy rules that have in their premises attributes of the classes.

In Section II we define the basic principles used for determining the importance of a class inside a software system and we present the global architecture of our tool. In Section III we describe how we extract the values of all class attributes used in the process of class ranking. Section IV presents how the raw values of attributes are preprocessed and transformed into fuzzy linguistic variables and defines the fuzzy rules for ranking classes according to their importance. Section V presents experimental results of applying our approach to a set of relevant open-source projects. Section VI discusses our results by comparing with related work. Section VII draws the conclusions of this paper.

II. Overview of our approach

A. Fundamental assumptions about the importance of classes

We consider that the importance of a class is given by the contribution it makes to the overall system: if a class is big and complex and has many interactions with other classes, it is a good candidate of being an important class of the system.

Also, previous works in this field introduced the idea that the importance of a class is also given by the importance of the classes it interacts with. To quantify this, the software system is modeled as a graph having the classes as nodes and their interactions as edges and network centrality metrics such as PageRank [3] are used.

In this work, we combine these two views, taking into account in the ranking process several class attributes, one of these attributes being the PageRank values of classes.

B. Architectural view of the class ranking tool

Figure 1 presents the global architecture of our tool.

The tool takes as input the code of the system to be analyzed. The raw values of the class attributes are extracted by static analysis of the source code with the help of the model extractors of the ART tool suite [4]. These raw values are normalized and scaled by a preprocessor. A Fuzzy Logic Controller (FLC) establishes the degree of importance of every class and is used to rank the classes according to their importance. The FLC's rules use the values of a class's

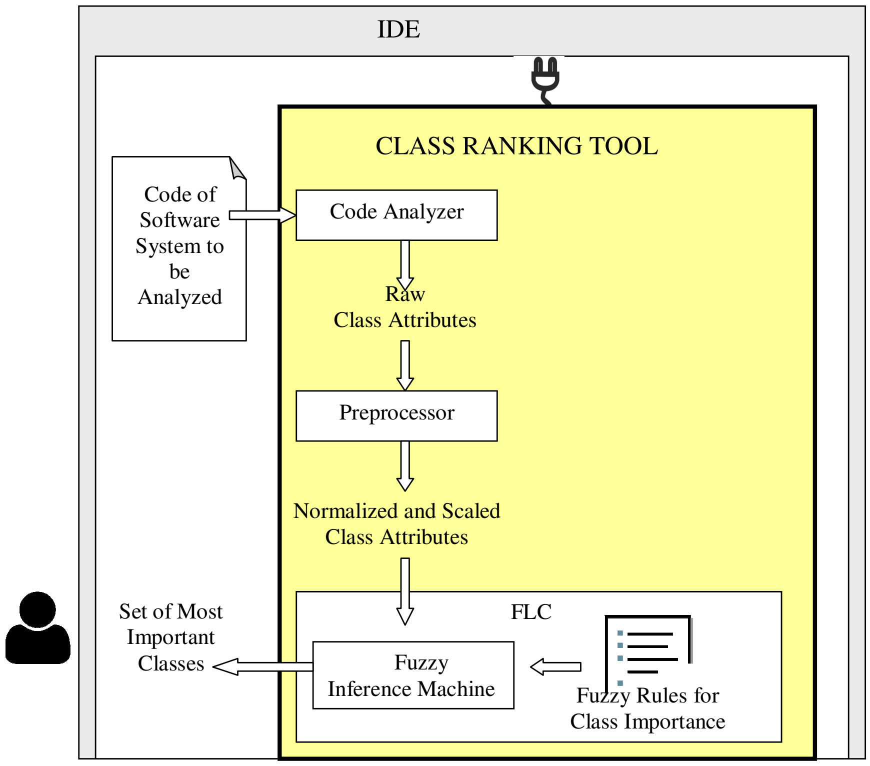
IDE
CLASS RANKING TOOL
Code of Software System to be Analyzed
Code Analyzer
Raw Class Attributes
Preprocessor
Normalized and Scaled Class Attributes
FLC
Set of Most Important Classes
Fuzzy Inference Machine
Fuzzy Rules for Class Importance

Fig. 1. Architecture and usage of the class ranking tool.

attributes to decide its suitability to belong in the set of key classes. A threshold is set for returning the top ranked classes which are the recommended set of key classes.

III. Extracting class attributes needed as inputs for ranking

The class attributes that are used as inputs are:

- Size: this is given by the number of methods of the class;
- WID (Weight of Incoming Dependencies)
- WOD (Weight of Outgoing Dependencies)
- PR (PageRank value of the class)

The weighting of dependencies and computation of PageRank values are briefly described in following subsections.

A. Weighting static dependencies between classes

Static dependencies in object oriented languages are produced by various situations: inheritance, realization, aggregation, field access, method call, parameter type, return type, local variable, type cast. A class A can depend on a class B by several situations of dependencies at the same time: for example, the class A can have members of type B, but in the same time it can have a method with parameters of type B and overall it can call several different methods of B.

The dependency relationship from a class A to a class B is characterized by a global *dependency strength*, which takes into account both the number and the kinds of dependency situations between the two classes. We compute the global dependency strength using a summative approach based on weights which reflect the ordering of dependency kinds according to their relative importance. Establishing these weights (the relative importance hierarchy of static dependencies kinds) is a subject of empirical estimation [5]. In this work, we use the order of relative importance for the different dependency kinds defined in [6].

The Weight of Incoming Dependencies (WID) for a class B is computed as the sum of the dependency strengths of all dependency relationships from a class A to B, for all classes A that depend on B.

The Weight of Outgoing Dependencies (WOD) for a class A is computed as the sum of the dependency strengths of all dependency relationships from A to a class B, for all classes B that class A depends upon.

B. Computation of PageRank values for Classes

PageRank [3] is a graph-based ranking algorithm which is wellknown due to its major 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 for that other node. Not all recommendations are equally valuable: the importance of the node giving the recommendation determines how important the recommendation itself is. It results that the score associated with a node, reflecting its importance, is given by both the recommendations that it receives and the scores of the nodes giving these recommendations.

In our previous work [7], [8] on using PageRank for identifying the key classes, the software system is modeled as a graph having as nodes classes or interfaces. In this model, the edges correspond to recommendations that derive from the program dependencies. If A depends on B, this means both that A gives a recommendation to B (called *forward recommendation*) but also that B gives a recommendation to A (called *back recommendation*). 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=1/2 of the weight of the forward recommendation from A to B, as it has been established in [8], [7]. In this work, we use the PageRank values computed in this way as PR attributes for the classes of the system.

IV. Fuzzy rules for computing the importance of a class

A. Linguistic variables

The class attributes that can be used as inputs are: Size, WID (Weight of Incoming dependencies), WOD (Weight of Outgoing Dependencies) and PR (PageRank value of class in graph of static class dependencies). Their raw values are obtained as presented in the previous section.

The range of the values of attributes depends on the size and type of the analyzed system, thus it is not possible to define in an absolute way, for example, when the size of a class is large: this will depend on the number of classes in the system and the average size of the classes in that system. A normalization of the values is needed in order to define the domain of the linguistic variable representing the class attribute. This has to take into account that in each system it is possible that a few classes have very extreme values. The range of values is thus limited to an area of double the standard deviation around the average value, which is considered the maximum value. Values that fall outside this area are assimilated with the maximum value (and they will correspond with very large in fuzzy terms). The values are then normalized and scaled to interval [0..100].

Each one of the four class attributes corresponds to a linguistic variable that is described as set of fuzzy terms. The terms have to be enumerated, for each term providing its name and the shape of its membership function. The implementation of the currently used FLC requires that the terms are of trapezoidal shape. The trapezoidal fuzzy sets are represented as quadruplets (a, b, c, d), where $0 \leq a \leq b \leq c \leq d$ are the x-coordinates of the trapeze points.

Choosing the number of fuzzy terms for each linguistic variable and the coordinates values of each term has been done by taking into account the importance of the corresponding class attribute in the decision process and the sensitivity of the decision to variations of the attribute. Taking these considerations into account, and after several rounds of tuning experiments, we defined the following terms of the linguistic variables:

Size={S, M, L}, where: S=(0, 0, 10, 40), M=(10, 40, 70, 90), L=(70, 95, 100, 100).

WID={VS, S, L, VL }, where: VS=(0, 0, 5, 10), S=(5, 10, 40, 60) L=(40, 60, 85, 98), VL= (85, 98, 100, 100).

WOD={VS, S, L, VL }, where: VS=(0, 0, 5, 10), S=(5, 10, 40, 60), L= (40, 60, 85, 98), VL= (85, 98, 100, 100).

PR={NL, L}, where: NL=(0, 0, 70, 90), L=(70, 90, 100, 100).

The most important attributes are WID and WOD, thus their linguistic variables have four terms, corresponding to very small (VS), small (S), large (L) and very large (VL). The shapes of VS and VL are narrower because, firstly, only extreme values must belong to these categories, and secondly, a number of extreme values of the raw attributes, that exceeded the average with more than the standard deviation were limited to the maximum value of 100. The PR attribute has been given only two terms, not large (NL) and large (L) because only large values of this attribute impact on the decision in the conclusion.

B. Fuzzy rules

We define fuzzy rules that have as inputs the attributes of a class and as conclusion the decision to select the class in the set of important classes. The linguistic variable Decision is defined with following terms corresponding to strong reject, weak reject, weak select, strong select: Decision={SR, WR, WS, SS}. The terms SR, WR, WS and SS are singletons with the coordinates evenly distributed over the domain of the output.

We experiment with two settings: first we use only three inputs given by the variables Size, WID, and WOD, and then add also the variable PR to the inputs.

1) Fuzzy rules with 3 inputs: The decision is taken using as inputs only the variables Size, WID, and WOD. We have 48 rules. An important class is generally characterized by big values of its attributes. However, not all attributes are equally important: a class with a big size but without interactions with other classes, shown by small weights of dependencies, will be probably a not important class. Also, it is possible that a small class with very strong interactions with other classes, shown by very large weights of the dependencies, is important. Regarding the interactions with other classes, it is a little bit more important to have many incoming dependencies than outgoing dependencies. These empirical observations are reflected in the set of rules. Since the limited space does not allow to list all the 48 fuzzy rules that result from all possible combinations of the inputs with 3, 4 and 4 terms, we illustrate with one rule given as example:

if Size=L and WID=S and WOD=L then Decision=WS

2) Fuzzy rules with 4 inputs: The decision is taken using as inputs the variables Size, WID, WOD and PR. We have 96 rules. The rules described in the subsection above do not distinguish the case when a somewhat not so large class, with not so large dependencies, is still important because its interactions are with very important classes. In order to correct this, we add the PageRank value of the class as an input to the rules. The rules described in the previous subsection are changed such that when PR is large then the value of the decision in conclusion is "incremented" with one term (e.g., if it was SR then it will be WR, etc). When PR is not large, then the decision is the same as in the case with three inputs. The rule given above as an example for FLC with 3 inputs is replaced by the following 2 rules in the case of FLC with 4 inputs:

if Size=L and WID=S and WOD=L and PR=L then Decision=SS

if Size=L and WID=S and WOD=L and PR=NL then Decision=WS

V. Experimental validation

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.

We run 3 methods of ranking:

Method 1: Use FLC with 3 inputs (Size, WID, WOD).

Method 2: Use PageRank values only, as in our previous approach described in [7] and [8].

Method 3: Use FLC with 4 inputs (Size, WID, WOD, PR).

B. Detailed analysis of experiments done on first case study

A first case study is JHotDraw[1], an open source customizable graphic framework for structured drawing editors. This system is also a much used benchmark for studies of empirical software engineering, being a well-known member of the Qualitas Corpus - a curated collection of software systems intended to be used for empirical studies on software code. We analyze here release 6.0b.1, its core of 398 classes. We consider that the set of key classes of JHotDraw is formed by these pointed out by the developers, completed with some classes added in the study of [9]: `Figure`, `Drawing`, `DrawingView`, `DrawApplication`, `Tool`, `Handle`, `DrawingEditor`, `StandardDrawingView`, `CompositeFigure`. This set of 9 classes is further considered the reference solution.

By applying the 3 ranking methods we obtain different ranking hierarchies. The classes occupying the top 30 positions for each method are illustrated in Figure 2. The classes from the reference solution set are highlighted.

Since there is no ranking specified across the classes in the reference set (and in most cases it is also not reasonable nor meaningful to expect such a ranking), we will record only the facts that the tool manages to find a class from the reference solution set on a certain top position. Ideally, if the reference solution set has K classes, the tool should rank them on the top K positions. The results in Figure 2 can be represented in a simplified way as in Figure 3. If we compare the 3 methods from the point of view of their results, we look at the number of hits in the Top 10, in the Top 20 and respectively Top 30. In the Top 10, method 1 finds 6 classes, while method 2 and method 3 find 5 classes. In the Top 20, method 3 finds all the 9 classes of the reference solution set, while method 1 and method 2 find only 8 classes. For this case study, the experiment shows that method 3 brings the best result, followed by method 1 and then by method 2.

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

C. Experimental results on other case studies

In addition to the first case study discussed in detail in previous subsection, we chose as case studies other four open source real life systems, which also have their design documentation available: Wro4J [2] (release 1.6.3), jEdit[3] (release 5.1.0), ArgoUML[4] (release 0.9.5) and JMeter[5] (version 2.0.1).

Table I summarizes the results obtained by applying the 3 ranking methods to the full set of case studies represented by all the five open source systems.

The size of the analyzed systems is measured in number of classes and ranges from 280 classes to 1266 classes. The reference solution (sets of key classes known from the design documentation of the systems) always contains about 10 classes (the numbers vary between 7 and 13).

We observe that method 3 (using FLC with 4 inputs, one of the inputs being PR) gives the best results for three of the case studies: JHotdraw, Wro4J and JEdit. For two of the case studies, JMeter and ArgoUML method 3 produces results similar to method 2. We can conclude that method 3 is globally better than method 2 (using PageRank values only). Although method 1 (using FLC with 3 inputs) happens to behave well on a few cases, overall it is outranked by the other two methods. The conclusion of our experiments is that the PageRank attribute of a class is an essential attribute for obtaining a good ranking solution, but taking into account additional class attributes can further improve the solution. The runtime for the complete end-to-end processing (from loading the code until obtaining the recommend set of important classes) is, even for large systems, not longer than 2-3 minutes.

VI. Discussion and Related Work

Different related works investigate methods for finding the most important classes from a software system, and their particularities concern following aspects: the primary information that is extracted and analyzed, the criteria used to define the importance of a class and the techniques used to identify the key classes.

The primary information is extracted in most cases by using static analysis of the code [10], [11], while a few approaches are using dynamic analysis [12], [13].

The criteria used to define the importance of a class are in most cases associating importance with interaction. The interactions of a class are measured either by design metrics such as coupling or by network metrics of the topology of the

[2]https://code.google.com/p/wro4j/

[3]http://jedit.org/

[4]http://argouml.tigris.org

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

Ranking position	FLC - 3 inputs .	PageRank only .	FLC - 4 inputs .
1	Figure	Figure	Figure
2	DrawingView	DrawingView	DrawingView
3	DrawingEditor	FigureEnumeration	FigureEnumeration
4	StorableInput	DrawingEditor	DrawingEditor
5	Drawing	Undoable	Undoable
6	StorableOutput	DrawApplication	DrawApplication
7	ConnectionFigure	StorableInput	StorableInput
8	Tool	Drawing	Drawing
9	Handle	StorableOutput	StorableOutput
10	UndoableAdapter	CollectionsFactory	CollectionsFactory
11	RelativeLocator	StandardDrawingView	StandardDrawingView
12	Undoable	ConnectionFigure	ConnectionFigure
13	AbstractCommand	AbstractCommand	AbstractCommand
14	Connector	DrawApplet	DrawApplet
15	TextFigure	CompositeFigure	Tool
16	FigureEnumerator	Command	Handle
17	FigureEnumeration	AbstractTool	UndoableAdapter
18	DrawApplication	Tool	RelativeLocator
19	CollectionsFactory	Connector	CompositeFigure
20	StandardDrawingView	TextFigure	AbstractTool
21	DrawApplet	HTMLTextAreaFigure	Connector
22	CompositeFigure	Locator	TextFigure
23	AbstractTool	PolyLineFigure	FigureEnumerator
24	HTMLTextAreaFigure	FigureChangeListener	HTMLTextAreaFigure
25	Locator	ConnectionTool	Locator
26	PolyLineFigure	Handle	PolyLineFigure
27	ConnectionTool	FigureAttributeConstant	ConnectionTool
28	HandleEnumeration	HandleEnumeration	HandleEnumeration
29	Storable	UndoableAdapter	Storable
30	DecoratorFigure	Storable	DecoratorFigure

Fig. 2. Top 30 ranked classes of JHotDraw.

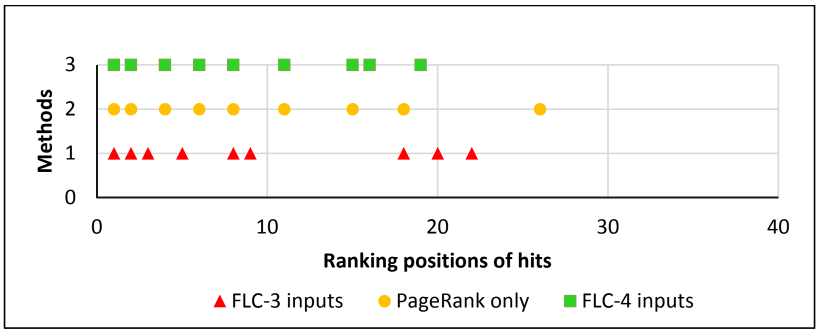

Fig. 3. Experiments with retrieving key classes of JHotDraw.

class interaction graph. Other approaches use information from the versioning system [14], considering that classes that are frequently changed are important. Further, textual information such as class names [15] can be used as a source of additional information for the importance of a class.

The techniques for identifying the key classes are mostly based on network analysis [11], [16], [17], and machine learning [10], [18].

It is difficult to compare the results of all these approaches, because they are using different software systems as case studies and not all publications describe the raw data of their experiments, such as the rankings that they obtained. Where such data was available we compared the results with our results for the same systems. In [8] and [7] we have compared in detail the results of our previous approach based only on PageRank (referred in this work as method 2) with the results of other works and showed that it produced solutions of similar quality as the state of the art, with significantly better runtime and less effort for the user.

In this work we have shown that by using fuzzy rules

TABLE I
EXPERIMENTAL RESULTS SUMMARY

	System	JHotDraw	Wro4J	jEdit	ArgoUML	JMeter
	System size	398	335	1266	852	280
	Reference set	9	12	7	12	13
Hits in Top 10	Method 1	6	5	5	3	2
	Method 2	5	6	4	6	4
	Method 3	5	6	4	6	4
Hits in Top 20	Method 1	8	10	5	5	5
	Method 2	8	9	6	8	7
	Method 3	9	10	6	7	7
Hits in Top 30	Method 1	9	12	6	7	6
	Method 2	9	11	6	9	8
	Method 3	9	12	6	8	8
Hits in Top 50	Method 1	9	12	7	8	8
	Method 2	9	12	6	10	8
	Method 3	9	12	7	10	8

with several inputs, including PageRank (corresponding with method 3 in this work), we can further improve the quality of the solution, doing better than our approach based on PageRank only, without increasing the overhead in runtime or user effort. Future work will try to improve this result by taking into account also other characteristics such as the system size and kind, and investigate if we can identify categories of systems where there is a specific set of rules that works best for each category.

VII. CONCLUSIONS

In this paper we propose a recommender tool to assist software engineers in tasks related to early stages of program comprehension, by finding the key classes of a software system. The experiments show that using fuzzy rules having as inputs class attributes such as size, weighted incoming dependencies, weighted outgoing dependencies and PageRank value produces the best results.

REFERENCES

[1] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *Software, IEEE*, vol. 27, no. 4, pp. 80–86, July 2010.

[2] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014.

[3] 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/

[4] I. Sora, "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.

[5] 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.

[6] I. Sora, G. Glodean, and M. Gligor, "Software architecture reconstruction: An approach based on combining graph clustering and partitioning," in *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, May 2010, pp. 259–264.

[7] I. Sora, "Finding the right needles in hay: Helping program comprehension of large software systems," in *2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, April 2015, pp. 129–140.

[8] I. Sora, "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)*, May 2015, pp. 495–500.

[9] 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.

[10] 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.

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

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

[13] M. Kamran, F. Azam, and A. Khanum, "Discovering core architecture classes to assist initial program comprehension," in *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, ser. Lecture Notes in Electrical Engineering, W. Lu, G. Cai, W. Liu, and W. Xing, Eds. Springer Berlin Heidelberg, 2013, vol. 211, pp. 3–10.

[14] M. Hammad, M. Collard, and J. Maletic, "Measuring class importance in the context of design evolution," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, June 2010, pp. 148–151.

[15] M. Osman, M. Chaudron, P. Van Der Putten, and T. Ho-Quang, "Condensing reverse engineered class diagrams through class name based abstraction," in *Information and Communication Technologies (WICT), 2014 Fourth World Congress on*, Dec 2014, pp. 158–163.

[16] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol, "Not all classes are created equal: toward a recommendation system for focusing testing," in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*. ACM, 2008, pp. 6–10.

[17] 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.

[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.