Source Code Quality Evaluation Using Network Science

Ivan Blažić, Marko Mišić, Zaharije Radivojević

***Abstract*— Structural quality of the source code has a great impact on the efficiency of software development processes. The cost and the complexity of adding functionalities or fixing a bug depend on how well-structured the code is. A standard way of measuring the structural quality of software is the calculation of code quality metrics, which are statistical data obtained by static source code analysis. This paper presents a software system for calculating code quality metrics by analyzing the network of dependencies between source code elements. Adaptations of well-known metrics can be calculated using this method, but with a more advanced application of network science new metrics can be introduced. Description of the applied method, the implemented system, as well as the results of a use case analysis are presented.**

***Index Terms*—Code quality metrics, network science, static source code analysis.**

I. Introduction

Code complexity rapidly increases the cost of performing changes to a software system, but also the risk of introducing bugs and errors [1]. For this reason, code maintainability becomes an important aspect of the software development process as a way of overcoming complexity issues. Maintainability is related to the effort needed to analyze, modify, or test a software system. It is usually achieved with a proper architecture and good coding practices that keep the components as simple and independent as possible [2]. This property is strongly affected by the structural quality of the software system.

Evaluation of the structural quality is usually done by calculating the code quality metrics, a collection of statistical data about the code structure [3]. Calculation of these metrics is commonly performed by simple static code analysis, which mostly consists of counting the number of elements and specific relationships, such as lines of code, number of classes, or number of methods called from a certain class. Common approach to source code quality analysis consists of combining traditional metrics in order to form a quality evaluation system [4][5].

Another approach to such analysis is the application of network science, the study of complex networks [6]. Software systems consist of code elements (e.g. classes, functions, or variables) and their dependencies (e.g. calls, inheritance, or variable access) which can be represented as a complex network [7]. The main benefit of the proposed network-based approach to source code analysis is the focus on the architecture and global topological properties of a software system as a whole, rather than a summary of local features [8]. For example, metrics based on advanced network analysis presented in the section IV imply the application of complex networks and cannot be calculated with a traditional approach.

In this paper, an approach to calculating code quality metrics by analyzing different types of dependency networks generated from the source code, as well as a software system that implements such approach, are presented.

The paper is divided as follows. The following section presents related work on the topic. A more detailed description of the dependency networks and applied approach is given in section III. Section IV describes used metrices, while functionalities of the implemented system are described in Section V. The results of use case code analysis are discussed in Section VI. Conclusion and directions for future work are given in the final section.

II. Related Work

Analysis of software systems using complex networks is an ongoing topic in the research community. This section provides a brief overview of the network-based approaches to software analysis applied in the related research.

Common application of network science is the study of social networks. Social network measure of the influence of an individual researcher called *h*-index has served as an inspiration to a new class centrality metric that has been introduced in [7].

Network-based measures have been used to predict defects on a program class or module level in several research papers [9][10][11]. Results have shown correlation between network properties and the number of defects on a class or module level. In some cases, the network-based measures can outperform the standard ones.

Structural quality measures based on network analysis have been introduced in previous research [12][13][14]. Network properties such as degree distribution, clustering, average path length, or modularity have been used to identify component cohesion, stability, complexity, and modular structure.

The related research is focused on the theory behind the proposed metrics, but few of them presents the technical implementation.

III. Dependency Networks

Code quality evaluation process presented in this paper consists of generating 3 types of dependency networks that are used to calculate 15 metrics. Each metric is calculated by

Ivan Blažić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: blazic.ivan@outlook.com).

Marko Mišić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: marko.misic@etf.bg.ac.rs).

Zaharije Radivojević is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: zaki@etf.bg.ac.rs).

analyzing one or two types of these networks. Values of all metrics are used to calculate a numerical grade for different aspects of quality. Final quality grade is the average of all sub-grades for individual quality aspects. The implemented software system allows the user to configure the influence of the individual metric to the quality grade and define aspects of quality.

The reason for using multiple types of networks is to separate the information needed for calculating different metrics. For simplicity, the following constraints are introduced in our approach:

- Focus is strictly on the software written in object-oriented paradigm.
- Only the source code written in Java language is supported.
- Annotations are ignored.
- Method implementations within interfaces are ignored, as this is considered as an improper use of interfaces.
- Dependencies to external libraries are ignored to avoid incomplete dependency networks.
- Built-in types and classes of Java standard library are ignored, as they considered to be a part of the language.
- All information is obtained by direct syntax-based analysis of explicitly defined dependencies. Therefore, any dependencies created by dynamic properties such as Java reflection or polymorphism are ignored. All implicit declarations such as inherited members or default constructors are not modeled.

All used networks are in a form of a directed simple graph. Each node, no matter the network type, has the following attributes:

- Identifier used in the code (e.g. class, method, or variable name).
- Type of the node (class, interface, method, field, or enum definition).
- Access modifier (public, private, protected, or default).

Links are commonly defined by a source node, a destination node, weight, and dependency information. Since two nodes can have multiple types of dependencies, link attributes are used to store this information. Self-links do not exist, therefore the source and destination nodes are always different. Types of source code dependencies covered in the presented approach are:

- Field type dependency (a class depends on all classes that are types of its members)
- Implementation (a class depends all on interfaces it implements)
- Inheritance (a class depends on its parent class)
- Inner class definition (a class depends on all inner classes it defines)
- Instantiation (a class depends on all classes that are types of objects it instantiates)
- Method call (a class depends on all classes that define methods it calls)
- Field definition (a class depends on its member variables and constants)
- Method definition (a class depends on its methods)

Depending on the actual source code dependencies they model, used networks are divided in the following 3 types.

A. *Call graph*

This network represents the dependencies between methods created by nested calls. Nodes represent the methods, and links represent calls. Direction of the links indicate the call hierarchy. This graph correlates to all method calls in the system.

B. *Class structure graph*

This network consists of several weakly connected components that represent each class. These components model the membership of fields (methods, constants, variables, and inner classes) to a class. Nodes can represent a class (central node), or different types of class members (peripheral nodes). Links represent field membership or a field usage in a method. Weight of the links between the central class node and a method node is the cyclomatic complexity [15] of the method.

C. *Type dependency graph*

This is a network of dependencies of any kind between each type (class or interface) in the system. Nodes represent types, and links are created between classes that have at least one of these dependencies between them. Figure 1 shows one part of a sample type dependency network.

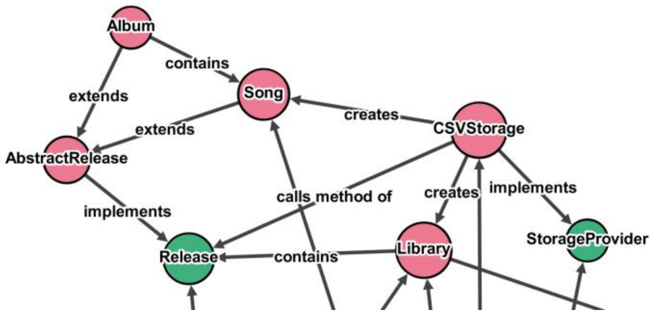

Fig. 1. Partial example of a type dependency network.

IV. Used metrics

This section provides an overview of code quality metrics used in the presented approach. All metrics used in this paper are based on network analysis. To have a better overview of the contribution that network science brings to software quality analysis, applied metrics are divided to the following two categories:

A. *Metrics based on simple network analysis*

This category includes the adaptations of a well-known set of metrics [3][15][16]. Adaptations are made for some metrics to fit the network-based approach. The use of networks in this category of metrics comes down to identifying and counting nodes and links with certain properties. This set includes the following metrics:

1) *Interface Size (IS)*

Total number of publicly accessible members of the class. This metric is a simple measure of the complexity of interfacing a class. It is calculated as the degree of the class node in the class structure graph, by observing only public members.

2) *Number of Methods (NOM)*

Total number of methods in a class, calculated as a class node degree in the class structure graph, by observing the method node neighbors. Methods with overloaded signatures are modeled as separate methods.

3) *Weighted Method Count (WMC)*

Sum of cyclomatic complexities of all methods in a class. Cyclomatic complexity is not calculated with a network-based approach, but with a simple abstract syntax tree traversal and count of branching elements. The value of method complexity is embedded as the weight of the link between a class node and a method node. Therefore, this metric is calculated in the same way as NOM, but with using a weighted degree instead.

4) *Response for Class (RFC)*

Sum of the number of public methods and the number of methods called from those methods. This is calculated by identifying the public method nodes in the class structure graph and summarizing their count with their degree in the call graph.

5) *Number of Implemented Interfaces (NII)*

This is an adaptation of *Number of Superclasses* metric. Since Java does not support multiple inheritance, this metric can only be applied to implementing interfaces. Calculated as the degree of a class node in the type dependency graph, by observing only links that represent the implementation dependency.

6) *Coupling Between Objects (CBO)*

Total number of other classes a class is coupled with. Coupling in this context means having any kind of dependency besides inheritance between two classes. Calculated as the degree of a class node in the type dependency graph, with filtering out the implementation and inheritance dependency.

7) *Tight Class Cohesion (TCC)*

This is a class modularity measure that evaluates the degree of member re-use. It is the ratio between the number of pairs of methods that commonly use at least one field variable and the total number of pairs of methods in a class. Pairs of methods are identified by searching the class structure graph for method definition and field usage links, the rest of the calculation is a simple formula for this metric.

8) *Number of Variable Fields (NOVF)*

Total number of member variables declared in a class. Calculated as a degree of the class node in the class structure graph, by observing only the variable member nodes as neighbors.

9) *Number of Subclasses (NSUB)*

Total number of classes inherited from the observed class or interface. Calculated as the in-degree of the node in the inheritance graph.

B. *Metrics based on advanced network analysis*

This category represents a set of metrics that are based on network properties [7][17] of three modelled networks, such as betweenness centrality of the nodes, network path lengths, cycle count, etc. A network-based approach to calculating these metrics is either required or more convenient due to their definition. Metrics from this category rely on complex network properties instead of simple link or node counts. These are the following metrics:

1) *Depth of Inheritance Tree (DIT)*

The longest inheritance path in the system. A longest path in a tree is its depth. Inheritance graph is created by selecting inheritance and implementation links from the type dependency graph. As the inheritance graph consists of multiple trees, the value of this metric is the maximum of depths of all trees.

2) *Number of Circularly Dependent Classes (NCDC)*

Total number of classes that are involved in at least one circular dependency. Calculated by identifying all cycles in the type dependency graph and counting the number of classes involved in those classes.

3) *Number of Disconnected Groups (NDG)*

Total number of groups of classes that are disconnected from other groups in the system. In terms of network science, this metric is the number of weakly connected components of the type dependency graph. Ideally, the whole network is weakly connected and there are no disconnected groups. Calculated by evaluating the connectivity of the type dependency graph and counting the number of weakly connected components.

4) *Degree of Interdependency (DOI)*

Ratio between the total number of existing dependencies between classes and interfaces in the system and the theoretical maximum number of those dependencies. This metric measures the total density of dependencies between all classes and interfaces. Calculated as the density of type dependency graph.

5) *Maximal Call Indirection (MCI)*

Length of the longest call path in the system. Calculated as the longest path in the call graph.

6) *Decoupling Impact (DI)*

Ratio between the number of type dependency paths that pass through the observed class and the total number of those paths. From a network science perspective, this is the betweenness centrality of class node in the type dependency graph. This metric measures the degree in which the node acts like a bridge between different groups of nodes.

From the quality analysis aspect, metrics are divided in class-level metrics (IS, NOM, WMC, RFC, NII, CBO, TCC, NOVF, NSUB and DI) and system-level metrics (DIT, NCDC, NDG, DOI and MCI). All metrics except TCC and DI represent a negative measure of quality, which means that higher quality structures should have as low values of these metrics as possible. The reason for this is that they all measure complexity and size, which are not desirable properties of software components. High cohesion on the other hand is desirable, as it measures modularity and re-use. The impact of the value of DI metric on quality can not be generalized, because it depends on the expected role of the observed class. Observations made regarding the use of this metric shall be presented in the results.

V. Software system implementation

A software system that applies the described approach was implemented within this paper [18]. It is a configurable console application for Java 1.8 platform. Configuration of the system allows the user to define various analysis parameters by simply editing JSON files available within the system. If the user does not change any parameters, the default configuration remains in use. With the system configuration the user is free to decide how the metric values influence the quality grade and which metric values are considered ideal. The system reads the configuration files, takes a source code directory as input, and performs the following functionalities:

A. Generating and exporting dependency networks

All dependency networks described previously are generated based on the source code given as an input. One additional network is generated, which is the mix of all networks called *unified graph*. This network is not used for the purpose of quality evaluation, but for a visual representation of all dependencies. These networks are exported as CSV, DOT, GEXF or GML files and can be used for a custom user analysis of the networks using other network analysis tools and libraries.

B. Quality grade evaluation

A grading system was introduced to present the quality evaluation results more intuitively. The user defines a reference value *Mref* for each metric. A metric grade *Gm* is calculated based on the reference value and the real metric value *M* as shown in (1) and represents a quality grade from a single metric perspective. TCC metric is an exception where *Gm* value is always the actual value.

A quality property grade *P* is defined by a name and a weighted sum of selected metric grades. For example, “Architecture” property grade can be influenced by NCDC, NDG, DOI and DIT with respective weights. The weight of each metric grade for a specific property is defined by the user as the metric factor *F*.

Actual quality grades range from 0 to *Gmax* defined by the user. Quality grade for a single property is calculated as shown in (2), and the final quality grade is the average of all quality property grades.

$$G_m = \begin{cases} \left(\frac{M_{ref}}{M}\right), & M > M_{ref} \\ 1, & M \leq M_{ref} \end{cases} \tag{1}$$

$$P = \sum_{i=1}^{N} (F(i) * G_m(i)) * G_{max} \tag{2}$$

C. Generating a quality analysis report

The results of the quality evaluation, based on the selected metrics, are presented in a form of a HTML report. This report contains the values of all metrics together with all grades and some additional information regarding the results (e.g. names of the classes involved in a circular dependency or names of the methods that make the longest call chain).

The system works by generating an abstract syntax tree (AST) of the source code, which is then traversed in order to create an abstract model of the code that later used when generating graphs, as shown on Figure 2. Each type of graph is created separately by iterating through the code model, filtering the elements and dependencies that are of interest for a specific graph, and converting this data into nodes and links. For example, the inheritance graph is created by extracting code elements that are represent classes or interfaces, and dependencies that represent inheritance or implementation. *Spoon* [19] library was used for generating the AST, and *JGraphT* [20] for working with graphs and networks.

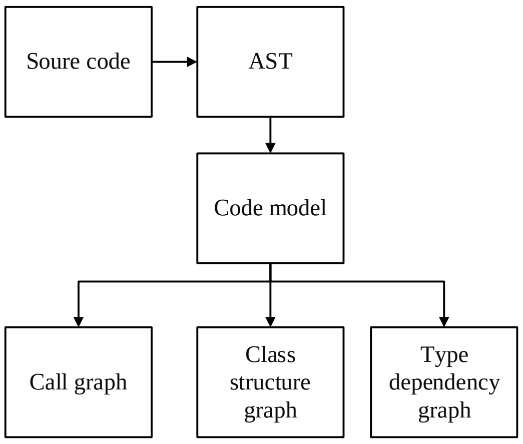

Fig. 2. Process of generating dependency networks from the source code.

VI. Results

All metrics covered in this paper are successfully implemented with the presented network-based approach. Each of the implemented metrics are covered with several automated test cases to verify their accuracy. This section presents a use-case of the approach, where the source code of the implemented software system is analyzed. Metrics that had the highest impact on the quality grade are presented, as well as the interpretation of DI metric. A more advanced analysis has been done for the exported type dependency network, and the results of this analysis are presented and discussed at the end of this section.

A. Analysis of metric results

Default system configuration was used in the presented use-case. In case of the source code of the implemented system NCDC, MCI and NDG metric values had the greatest impact on the quality grade. Table 1 presents the metric grades *Gm* for each system-level metric, as well as their actual and reference values. As described in section V, a metric grade is calculated based on actual and reference value of a metric. Reference values are specified in the system configuration. Class-level metrics did not have a significant impact on the quality grade.

TABLE I
VALUES OF SYSTEM-LEVEL METRICS

Metric	Reference value (Mref)	Actual value (M)	Metric grade (Gm)
NCDC	0	30	0
MCI	4	10	0,4
DIT	5	2	1
DOI	0.3	0.025	1
NDG	0	2	0

B. Interpretation of DI metric

The impact of betweenness centrality on the structural quality of the source code cannot be generalized. This metric is a measure of separation between distinct groups of nodes. Depending on the predicted position of the node (class or interface), a high value of this metric may or may not be desirable. For example, utility classes that are accessed from many different parts of the code or interfaces that are intended to separate two bigger components are expected to have a high DI. Since these are special architectural roles, in most cases high DI values are not desirable.

Several classes in the implemented system have unexpectedly high DI, twice the average value or more. The cause of such values is the direct dependency to classes that normally have a high DI. In this way the observed class also has a high DI due to transitive dependencies inherited from another node.

C. Advanced analysis of the type dependency network

Properties of the exported network of type dependencies have been additionally analyzed outside of the implemented software tool. This analysis was done with Gephi [21] with the goal to study the community structure of the network as well as to examine the real-world properties.

1) Community structure

Community detection algorithm with default parameters of Gephi was applied to the network. Visualization of the results is shown in Figure 3, where a strong community structure can be observed. Nodes are grouped and colored by their association to communities.

The detected communities strongly resemble the relationship between classes in the same package or functionality. Amount of correlation between the community structure and source code module separation can indicate architectural flaws. Two examples are observed in the given network.

A certain node has been identified to strongly belong in the community of another package. This can be caused either by the lack of an appropriate interface or a mistake in functional decomposition.

Software modules should be strongly decoupled, which should reflect in strongly separated communities. Not all communities in the presented network are clearly separated, which indicates weak module decoupling.

2) Real world properties

Networks that model real systems are expected to have special properties comparing to ones that are random or trivial [22]. These properties include a strongly expressed community structure, a short average path length relative to the network size, and a *power-law* degree distribution that resembles the distinction of central and peripheral nodes that can be seen in preferential attachment network models.

Degree distribution of type dependency graph of the analyzed system is shown in Figure 4. Network diameter is 10, while the average path length is 4.03. All those properties of the presented network clearly show that it resembles real-world networks.

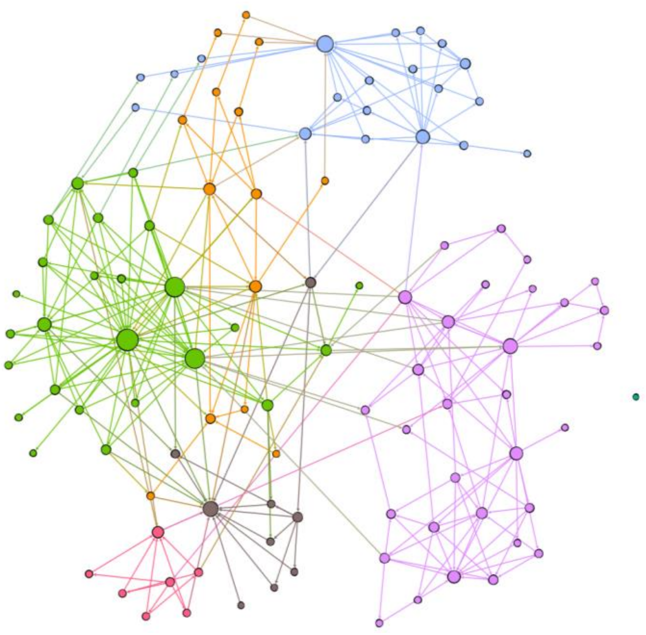

Fig. 3. Visual representation of the type dependency network of the implemented software system.

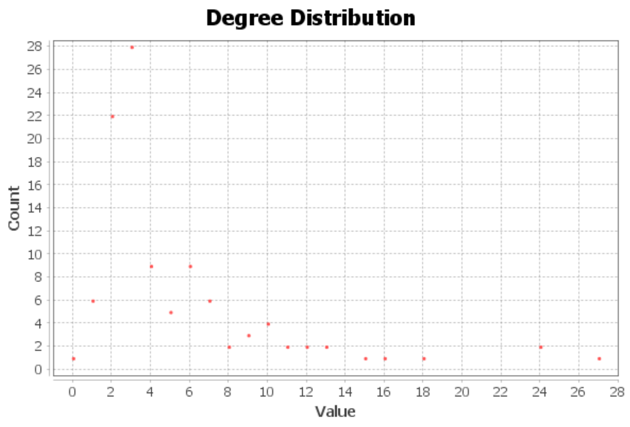

Fig. 4. Degree distribution of type dependency graph of analyzed system.

VII. Conclusion

This paper presents a software structure quality evaluation method based on the analysis of dependency networks generated from the source code. Three types of dependency networks were used to calculate a set of code quality metrics. Several well-known metrics have been adapted for this approach, and new network-based metrics were introduced. A software system that applies this approach has been implemented. A use-case of the implemented system, as well as an advanced analysis of one of the dependency networks are presented in the results.

Proposed approach has been successfully implemented and tested with the introduced software system. Additional examination of the properties of one generated network provided more information about architectural flaws and real-world properties of the source code. The downside of the proposed approach is the additional complexity of analyzing dependency networks that is not necessary for calculating simple metrics. This additional complexity also implies a lower calculation performance comparing to the traditional approach.

Implemented software system can be improved to support fine-grained dependency networks on the lexical level, such as control and data dependency graphs. Support for other programming languages would further improve the usability of the system. Topics of future research would include additional network-based metrics, such as class instability, clustering coefficient, and modularity. Another approach to future research would be the study of correlation between dependency network properties and reliable quality measurements.

ACKNOWLEDGMENT

This work has been partially funded by the Ministry of Education, Science, and Technological Development of the Republic of Serbia. Grant numbers III44009 and TR32047.

REFERENCES

[1] V. Antinyan, M. Staron, A. Sandberg, “Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time”, *Empirical Software Engineering*, vol. 22, no. 6, pp. 3057-3087, Mar, 2017.

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall, Upper Saddle River, New Jersey, US, 2017.

[3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design”, *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, Jun, 1994.

[4] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, H. Watanabe, “A Framework for Measuring and Evaluating Program Source Code Quality”, Proc. Product-Focused Software Process Improvement, 8th International Conference, PROFES, Riga, Latvia, vol. 4589, pp. 284-299, Jul. 2-4, 2007.

[5] F. Madou, M. Agüero, G. Esperón, D. López De Luise, “Software for Improving Source Code Quality”, *World Academy of Science, Engineering and Technology*, no. 59, pp. 1259-1265, 2011.

[6] National Research Council, *Network Science*, The National Academies Press, Washington, DC, US, 2005.

[7] Y. L. Ding, , B. He. Peng, “An Improved Approach to Identifying Key Classes in Weighted Software Network”, *Mathematical Problems in Engineering*, vol. 2016, pp. 1-9, Jan, 2016.

[8] W. Pan, “Applying complex network theory to software structure analysis”, *World Academy of Science, Engineering and Technology*, vol 60, pp 1636-1642, Dec, 2011.

[9] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, “An Empirical Study of Social Networks Metrics in Object-Oriented Software”, *Advances in Software Engineering*, vol. 2010, pp. 1-21, Jan, 2010.

[10] T. H. D. Nguyen, B. Adams, A. E. Hassan, "Studying the impact of dependency network measures on software quality", Proc. 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, pp. 1-10, Sep. 12-18, 2010.

[11] M. Orrù, C. Monni, M. Marchesi, G. Concas, R. Tonelli, “Predicting Software Defectiveness through Network Analysis”, Proc. Seminar On Advanced Techniques and Tools for Software Evolution, Mons, Belgium, vol. 1820, pp. 36-47, Jul. 6-8, 2010.

[12] M. Savić, Ivanović Mirjana, “Graph clustering evaluation metrics as software metrics”, Proc. 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications, Lovran, Croatia, vol. 1266, pp. 81-89, Sep. 19-22, 2014.

[13] L. Šubelj, M. Bajec, “Software systems through complex networks science: review, analysis and applications”, Proc. First International Workshop on Software Mining, Beijing, China, pp. 9-16, Dec., 2012.

[14] W. Pan, C. Chai, “Measuring software stability based on complex networks in software”, *Cluster Computing*, vol. 22, no. 2, pp. 2589-2598, Mar, 2019.

[15] T. J. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec., 1976.

[16] J. M. Bieman, B-K- Kang, “Cohesion and reuse in an object-oriented system”, *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 259-262, Aug, 1995.

[17] T. D. Oyetoyan, “Dependency cycles in software systems: quality issues and opportunities for refactoring.”, Ph.D. dissertation, NTNU, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science, Trondheim, Norway, 2015.

[18] I. Blažić, “Code Quality Analysis Using Network Science”, Source code, https://github.com/BlazicIvan/Net-CQA, 2020.

[19] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier, “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”, *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155-1179, Sep, 2016.

[20] D. Michail, J. Kinable, B. Naveh, J. V. Sichi, “JGraphT—A Java Library for Graph Data Structures and Algorithms”, *ACM Transactions of Mathematical Software*, vol. 46, no. 2, Article 16, Jun, 2020.

[21] M. Bastian, S. Heymann, M. Jacomy, “Gephi: an open source software for exploring and manipulating networks”, Proc. International AAAI Conference on Weblogs and Social Media, San Jose, California, May 17 - 20, 2009.

[22] R. Albert, A-L. Barabási, “Statistical Mechanics of Complex Networks”, *Reviews of Modern Physics*, vol 74, no. 1, pp 47–97, Jan, 2002.