

Software Systems through Complex Networks Science: Review, Analysis and Applications

Lovro Šubelj
University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
lovro.subelj@fri.uni-lj.si

Marko Bajec
University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
marko.bajec@fri.uni-lj.si

ABSTRACT

Complex software systems are among most sophisticated human-made systems, yet only little is known about the actual structure of 'good' software. We here study different software systems developed in Java from the perspective of network science. The study reveals that network theory can provide a prominent set of techniques for the exploratory analysis of large complex software system. We further identify several applications in software engineering, and propose different network-based quality indicators that address software design, efficiency, reusability, vulnerability, controllability and other. We also highlight various interesting findings, e.g., software systems are highly vulnerable to processes like bug propagation, however, they are not easily controllable.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures, software science*

General Terms

Theory, algorithms, experimentation.

Keywords

Software systems, Software engineering, Software networks, Network analysis.

1. INTRODUCTION

Complex software systems are among most sophisticated systems ever created by human. Nevertheless, only little is known about the actual structure and quantitative properties of large software systems [6]. For instance, in the context of software engineering, one is interested in how 'good' software looks like. Commonly adopted approaches and techniques fail to give a comprehensive answer [5, 7], moreover, there is also a lack of a simple but yet rigorous framework for

software analysis (to our knowledge). The above dilemma was denoted *software law* problem [6], which urges towards identifying (physical) laws obeyed by software systems that could be used in practical applications.

Networks possibly provide the most adequate framework for the analysis of the structure of complex systems like software projects¹. Also, due to their simple and intelligible form, analysis of different networks has already provided several significant discoveries in the last decade [46, 3, 16, 23]. Note that the adoption of software networks is not novel [35, 27, 19, 39], however, network analysis is still only rarely used in software engineering. The main purpose of this study is thus to highlight different techniques developed in the field of network analysis, and to expose their use in software comprehension, development and engineering. We review most of the past work on different types of software networks, whereas we also include network analysis techniques proposed just recently [23, 44]. (Note that the main focus of the paper is merely a review, rather than a detailed comparison of network analysis techniques with other approaches.)

The study in the paper analyses software networks on different levels of granularity. First, we address the macroscopic properties of software networks like scale-free and small-world phenomena [46, 3] that are related to the structure and design of the entire project, or projects, represented by the network. Second, we analyze the microscopic properties of individual nodes, with special emphasis on different dynamical processes occurring on software networks like bug propagation [2, 30]. The above can be related to software quality, complexity, reusability, robustness, vulnerability and controllability. Third, we also identify mesoscopic structural modules within software networks [16, 44] and show their applicability in the context of software abstraction and refactoring. The paper thus exposes network analysis as a prominent set of techniques for software engineering.

The rest of the paper is structured as follows. Section 2 introduces software networks used in the study. Section 3 analyzes different characteristics of adopted networks and discusses their use in software engineering. Some applications of the presented techniques are given in Section 4, while Section 5 concludes the paper.

2. SOFTWARE NETWORKS

Various types of networks have been proposed for the analysis of the structure of complex software systems. For

¹Throughout the paper, the term project refers to a repository of software code.

```

class A extends S implements I {
    F field;
    public A (P parameter) {
        ...
    }
    public R function(P parameter) {
        ...
        return R;
    }
}

```

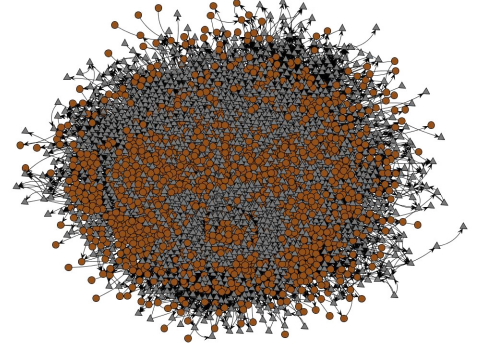
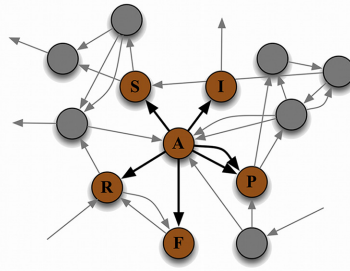


Figure 1: (left) A simple Java class and the corresponding part of class dependency network. Direction of links is (mostly) just the opposite to the flow of information. (right) Class dependency network of java (circles) and javax (triangles) namespaces of Java language.

Table 1: Properties of class dependency networks used in the study.

Network	Project	n	m	k	LCC	$ A $	$ P $
<i>flmng</i>	Flamingo 4.1 (GUI components) [39]	141	269	3.82	0.88	153	18
<i>colt</i>	Colt 1.2.0 (scientific computation) [39]	243	720	5.93	0.94	267	21
<i>jung</i>	JUNG 2.0.1 (network analysis) [39]	317	719	4.54	0.96	357	41
<i>org</i>	Java 1.6.0.7 (org namespace) [39]	709	3571	10.07	0.69	778	50
<i>weka</i>	Weka 3.6.6 (data mining framework)	953	4097	8.60	0.98	1054	84
<i>javax</i>	Java 1.6.0.7 (javax namespace) [39]	1595	5287	6.63	0.44	1889	118
<i>java</i>	Java 1.6.0.7 (java namespace) [39]	1516	10049	13.26	1.00	1518	56

instance, software architecture maps [35], software mirror graphs [6], class, method and package collaboration graphs [17], subroutine call graphs [27], inter-package dependency networks [21], software class diagrams [37] and class dependency networks [39], to name just a few. Networks mainly divide whether they are constructed from source code, byte code or program execution traces, and due to the level of software architecture represented by the nodes, and the set of interdependencies represented by the links.

For consistency with some previous work [4, 17, 47, 39], we construct networks from the source code of different Java projects² (Table 1). Due to the object-oriented view of Java language, nodes in the network can represent either project packages, software classes, methods and functions or individual lines of code. We here adopt class dependency networks [39], where nodes represent classes and links correspond to different dependencies among them (Figure 1). The latter is based on the following reasons. First, as networks are constructed merely from the signatures of different classes, and functions and fields therein, they are only mildly influenced by the subjective nature of each individual developer. (This can be more adequately modeled by, e.g., text mining applied to the names of different programming constructs [20].) Second, mesoscopic structures of class dependency networks coincide with project packages [39, 44], which enables various applications in software engineering (Section 4). Third, such networks relate to the information flow between different parts of software project, and also coincide with the human comprehension of the object-oriented software systems.

Note that class dependency networks address only the

inter-class structure of the software project, whereas the intra-class dependencies are disregarded. However, similarly as above, the latter reflect also the programming style of a particular developer, rather than the intrinsic structure of the software project alone. Nevertheless, future work will extend the study to inter- and intra-class dependencies using to concepts of interdependent or coupled networks [26, 15].

Formally, let a project consist of classes $A = \{A_1, A_2, \dots\}$ and let P be the set of software packages (bottom-most level of the package hierarchy). Corresponding class dependency network is then a directed graph $G(N, L)$, with nodes $N = \{1, \dots, n\}$ and links L ($m = |L|$). Node i corresponds to a class A_i , however, since isolated nodes are discarded in the analysis, $n \leq |A|$. A directed link $(i, j) \in L$ represent some dependency between classes A_i and A_j : inheritance (A_i inherits or implements A_j), parameter (A_i contains a method, function or constructor that takes A_j as parameter), return (A_i contains a method or function that returns A_j) and field (A_i contains a field of type A_j). Denote k to be the average degree in the network (i.e., average number of links incident to a node). Furthermore, let k^{in} and k^{out} be the average in-degree and out-degree of the nodes, $k = k^{in} + k^{out}$. Hence, k_i^{out} corresponds to a number of other classes required to implement the functionality of a respective class A_i , while k_i^{in} corresponds to the number of classes that use (depend on) A_i . Last, denote LCC to be the fraction of nodes in the largest connected component³.

Table 1 shows properties of class dependency networks used in the study. Networks were selected thus to represent a diverse set of software systems including utility libraries

²Networks are available from <http://lovro.lpt.fri.uni-lj.si/>.

³All networks in figures are reduced to LCC-s.

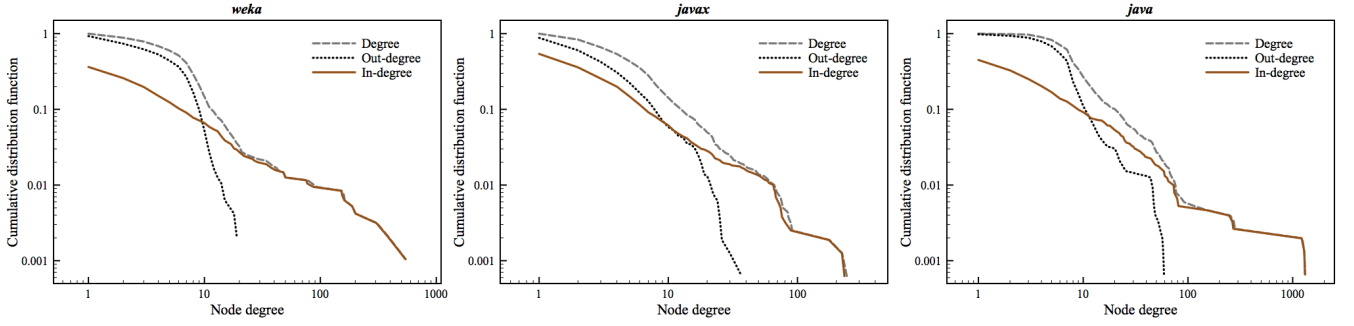


Figure 2: Degree distributions of *weka*, *javax* and *java* networks.

Table 2: Different statistics for class dependency networks used in the study.

Network	γ	C	D	C_{ER}	l	E	l_{ER}	n_d/n
<i>flmng</i>	3.0	0.25	0.31	0.03	4.05	0.03	3.47	0.38
<i>colt</i>	2.7	0.41	0.47	0.02	3.44	0.03	3.16	0.30
<i>jung</i>	2.5	0.37	0.42	0.01	4.19	0.02	3.88	0.48
<i>org</i>	2.2	0.57	0.62	0.01	2.68	0.03	2.81	0.39
<i>weka</i>	3.0	0.39	0.43	0.01	2.91	0.01	3.39	0.12
<i>javax</i>	2.6	0.38	0.44	0.00	3.88	0.02	3.16	0.30
<i>java</i>	2.4	0.69	0.73	0.01	2.18	0.02	3.09	0.17

(e.g., *flmng* and *colt* networks), complete frameworks (e.g., *jung* and *weka* networks) and also the core of Java language itself (i.e., *java* network).

Software networks are compared against Erdős-Rényi random graphs [12], where a link are placed between each pair of n nodes with probability $k/(n-1)$, where $k = 2m/n$ for some n and m .

3. ANALYSIS AND DISCUSSION

3.1 Scale-free networks – software complexity and reusability

Simple random graphs experience a Poisson degree distribution p_k , $p_k \sim \frac{\lambda^k e^{-\lambda}}{k!}$. On the contrary, p_k of most real-world networks including software networks follows a power-law form $p_k \sim k^{-\gamma}$ [3, 35, 31, 9], where γ is a scale-free exponent, $\gamma > 1$. The latter can be clearly observed by a straight line with slope $-\gamma$ in a log-log plot (Figure 2). Networks with power-law degree distributions are denoted scale-free, while γ can be directly related to the spreading processes occurring on networks [32] (e.g., bug propagation). For $\gamma \in (2, 3)$, even a very small fraction of faulty nodes can already render the entire system inapplicable [30, 32]. Unfortunately, the latter applies for all software networks used here (Table 2).

Scale-free networks are usually considered an artifact of Yule’s process or *rich get richer* phenomena [3]. For class dependency networks, this refers to the fact that highly used classes are, obviously, well known among developers, and would thus also be more commonly adopted in the future. However, power-laws should thus emerge merely in the in-degree distribution p_k^{in} that refers to the number of times each class is used [36, 4] (Figure 2). More precisely, scale-free nature of p_k^{in} is a result of high code reusability. On the other hand, out-degree distribution p_k^{out} is related to

software complexity, since classes with high k_i^{out} encompass most complex functionality. Here, complexity refers to the number of other classes needed to implement the functionality of the respective class. For example, most commonly reused class in *java* network is **String**, whereas **FileDialog** is the most complex one (Table 3).

Well developed software project should thus exhibit scale-free p_k^{in} and highly truncated p_k^{out} . Next, lower γ indicates higher code reuse, which also decreases the probability of fault propagation throughout the system. Last, classes with very high k_i^{out} , and also k_i^{in} , should be implemented with extra care (see Section 3.3).

3.2 Small-world networks – software structure and design

Software networks exhibit small-world phenomena [46] (see [27, 38] and Table 2), which refers to high clustering C [46] and very short average distance between the nodes l [1] (also known as six degrees of separation [25]). C measures transitivity in the network, and is defined as the probability that two neighbors of a node are also linked, $C \in [0, 1]$. $l = \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij}$, where d_{ij} is the distance between i in j in the respective undirected network (i.e., number of links in the shortest path). Small-world networks most commonly refer to $C \gg C_{ER}$ and $l \approx l_{ER}$ [46], where C_{ER} and l_{ER} are the values for a corresponding random graph.

Clustering of software networks can be related to intrinsic characteristics of the underlying systems [43]. For instance, visualization classes usually experience very high clustering, while clustering is almost zero for I/O classes [43, 40].

Average distance l is an important indicator of the structural design of the project, or projects, represented by the network. More precisely, since $l \approx l_{ER}$, $l \gg l_{ER}$ indicates that the underlying software system has divided into several independent parts with rather different functionality (Fig-

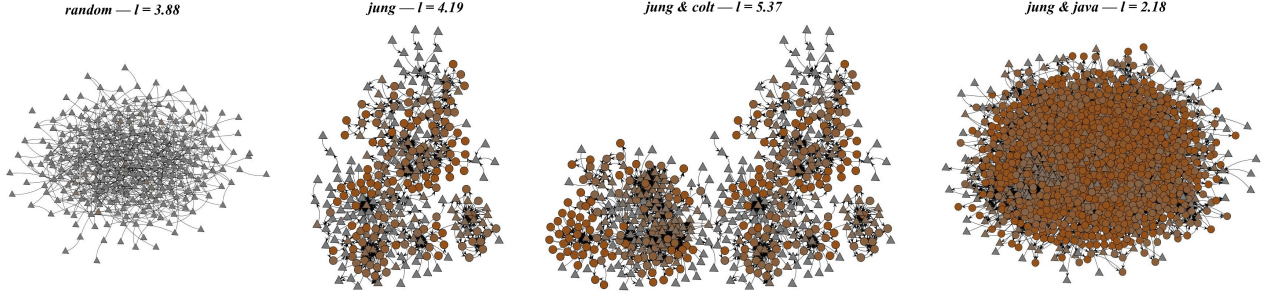


Figure 3: A random graph, *jung* network, *jung & colt* network and *jung & java* network. Average distance between the nodes l equals 3.88, 4.19, 5.37 and 2.18. Node symbols correspond to clustering D [33] that ranges between 0 (triangles) and 1 (circles).

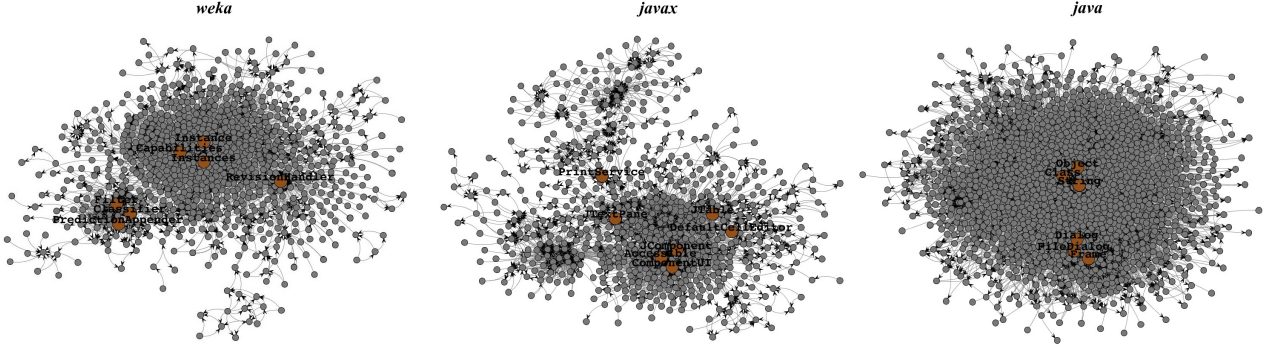


Figure 4: *weka*, *javax* and *java* networks with highlighted seed nodes.

ure 3). Note also that software networks should never be combined with the core of the language, since the latter completely obscures its structure and dynamics.

It ought to be mentioned that software networks are small-world only in the undirected case [19]. The contrary would imply a cyclic flow of information within the software project. For instance, high-level Java class `String` does not use the functionality of a lower-level `FileDialog`. Let E be the efficiency of network information flow [22] defined as $E = \frac{1}{n(n-1)} \sum_{i \neq j} 1/d'_{ij}$, where d'_{ij} is the distance from i to j in a (directed) network, $E \in [0, 1]$. Small-worlds should result in high flow efficiency E , however, software networks have $E \approx 0$ (Table 2).

Well designed software project should thus experience $C \gg C_{ER}$, $l \approx l_{ER}$ and $E \approx 0$. Also, one should be wary of $l \gg l_{ER}$ throughout the project development.

3.3 Network nodes – software vulnerability and control

In the context of spreading processes on software networks [28, 45] (e.g., bug propagation) and network robustness [2, 32] (i.e., software vulnerability), one is interested into so called seed nodes that could originate the propagation of faults through the entire system⁴. Centrality metrics that measure nodes influence are commonly regarded as a prominent indicator of seed nodes [13, 14]. Denote DC_i to be the degree centrality defined as $DC_i = k_i/(n-1)$,

⁴Although a poor implementation of any software class already makes the system vulnerable, the problem is even amplified in the case of, e.g., highly reused classes.

where k_i is the degree of node i , $DC_i \in [0, 1]$. Next, denote CC_i to be the harmonic closeness centrality defined as the average inverse of distance from i to the rest of the nodes, $CC_i = \frac{1}{n-1} \sum_{i \neq j} 1/d'_{ij}$, $CC_i \in [0, 1]$. Last, denote BC_i to be the betweenness centrality defined as the fraction of shortest paths between the nodes that go through i , $BC_i \in [0, 1]$.

As $k_i \approx k_i^{in}$ for software networks, DC_i actually identifies classes with the highest code reuse or, equivalently, high in-degree k_i^{in} (Table 3). Similar set of influential classes is reported by BC_i (Table 4). On the other hand, CC_i identifies classes that somewhat coincide with high complexity classes identified in Section 3.1. BC_i (and DC_i) thus reveals classes whose faulty implementation could influence the entire system, whereas CC_i exposes classes that are most prone to an arbitrary fault within the system. The former commonly reside in the core of the respective software network, while the latter are found in the periphery (Figure 4).

Extra care should be put in the development of classes with high BC_i , while high CC_i classes can be adopted for an effective, and also efficient, software testing.

Network controllability has just recently been proposed for the analysis of directed real-world networks [24, 23]. Here, one is particularly interested in the number of driver nodes n_d that one has to govern in order guide the entire system [23] (i.e., gain control over the output of the system under the assumption of simple linear transformations). For scale-free networks with p_k^{in} equal to p_k^{out} , $n_d/n \approx e^{k(\gamma-2)/(2-2\gamma)}$, $\gamma > 2$ [23]. Note that, contrary to seed nodes (Table 4) and general belief, driver nodes tend to avoid high degree nodes [23, 11].

Table 3: Hubs (i.e., nodes with very high degree) within *weka*, *javax* and *java* networks.

<i>weka</i>			<i>javax</i>			<i>java</i>		
Node	k_i^{in}	k_i^{out}	Node	k_i^{in}	k_i^{out}	Node	k_i^{in}	k_i^{out}
Instances	541	5	JComponent	235	11	String	1308	7
Instance	381	4	Accessible	222	1	Class	1288	4
Capabilities	304	4	ComponentUI	175	2	Object	1228	1
ClassAssigner	0	19	JTable	6	37	FileDialog	0	59
Filter	0	19	JTextPane	0	30	Frame	4	58
Classifier	0	18	JMenu	1	26	Dialog	5	57

Table 4: Seed nodes (i.e., very influential nodes) within *weka*, *javax* and *java* networks.

<i>weka</i>			<i>javax</i>			<i>java</i>		
Node	CC _i	BC _i	Node	CC _i	BC _i	Node	CC _i	BC _i
PredictionAppender	0.03	0.00	DefaultCellEditor	0.10	0.00	FileDialog	0.09	0.00
Classifier	0.03	0.01	JTable	0.10	0.12	Dialog	0.09	0.00
Filter	0.03	0.00	JTextPane	0.09	0.08	Frame	0.09	0.00
Instances	0.01	0.51	JComponent	0.04	0.23	String	0.02	0.36
RevisionHandler	0.00	0.26	Accessible	0.01	0.18	Object	0.02	0.32
Instance	0.01	0.13	PrintService	0.02	0.17	Class	0.02	0.26

Most software network are not highly controllable, since one would have to manage 30-50% of classes in order to control the entire project (Table 2). Nevertheless, due to high density, the core of Java language can be controlled through merely 17% of classes in *java* namespace. For comparison, n_d/n equals $\approx 80\%$ for regulatory networks, $\approx 50\%$ for the Internet, $\approx 30\%$ for power grids and on-line social networks, while, interestingly, it is below 3% for corporate ownership networks [23].

Controllability of a software system can be limited by decreasing k or γ , which is achieved by decreasing code complexity and increasing code reuse (Section 3.1).

3.4 Network modules – software aggregation and modularity

Packages of the software system reflect in different structural modules within class dependency networks [39, 44]. For instance, visualization classes commonly aggregate into communities of densely connected nodes [16], whereas different parsers, transformers or plugins often arrange into functional modules [43] that correspond to (disconnected) groups of nodes with common linkage patterns. Otherwise, clear community structure signifies highly modular structure of the respective software system, while well supported functional modules are related to clear functional roles of the classes within the project [39, 44, 43].

Table 5 compares software packages against network modules identified with MO [8] and CP [42, 41] community detection approaches, and MM [29] and GP [44, 43] structural module identification algorithms. Analysis reveals that general structural modules including communities and functional modules most accurately model the package structure of the software systems in this study.

4. APPLICATIONS

Due to space limitations, the following section only briefly describes different applications of network analysis techniques presented in Section 3. Future work will focus on a more detailed examination and development of supporting imple-

mentations that could be easily applied in practice.

4.1 Software project abstraction

Figure 5 shows an application of network structural module detection to software project abstraction. One can identify an entire hierarchy of modules that is consistent with the package hierarchy, while also enclosing class dependencies that go beyond packages decided by the developers. Besides better comprehension, revealed hierarchy enables the prediction of dependencies between the classes of a project [43].

4.2 Software packages refactoring

Network module detection algorithms can also be applied for refactoring of software packages [39, 43]. One can adopt a community detection algorithm to reveal highly modular structure (Figure 6, (left)) or a functional module detection algorithm to identify the underlying functional structure (Figure 6, (middle)). General structural module detection algorithms partition software classes according to both modular and functional links that are present among the dependencies of the project (Figure 6, (right)).

4.3 Software packages prediction

Table 6 shows classification accuracies for the prediction of software packages for the classes of different systems. Let i be a node corresponding to class A_i . Package of A_i is then predicted to be the most likely package considering nodes within the same structural module as i . The nodes are weighted according to Jaccard similarity [18], which is defined as $|\Gamma_i \cap \Gamma_j|/|\Gamma_i \cup \Gamma_j|$, where j is a similar node and Γ_i is the neighborhood of node i . Structural modules are identified with the algorithm in [44, 43].

On average, one can predict software packages with $\approx 80\%$ probability for most classes of the systems considered, whereas complete package hierarchy can be precisely identified for over 60% of the software classes (Table 6).

4.4 Software quality indicators

Table 7 and Table 8 show software project and class qual-

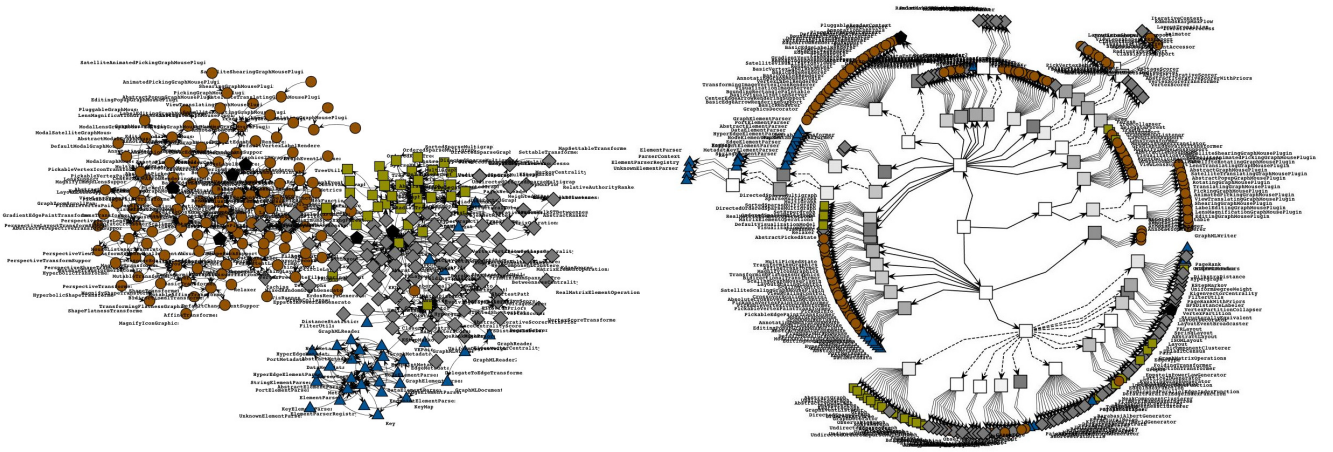


Figure 5: (left) *jung* network where node symbols represent high-level packages of JUNG framework: visualization (circles), io (triangles), graph (squares) and algorithms (diamonds). (right) Hierarchy of structural modules revealed with the algorithm in [43].

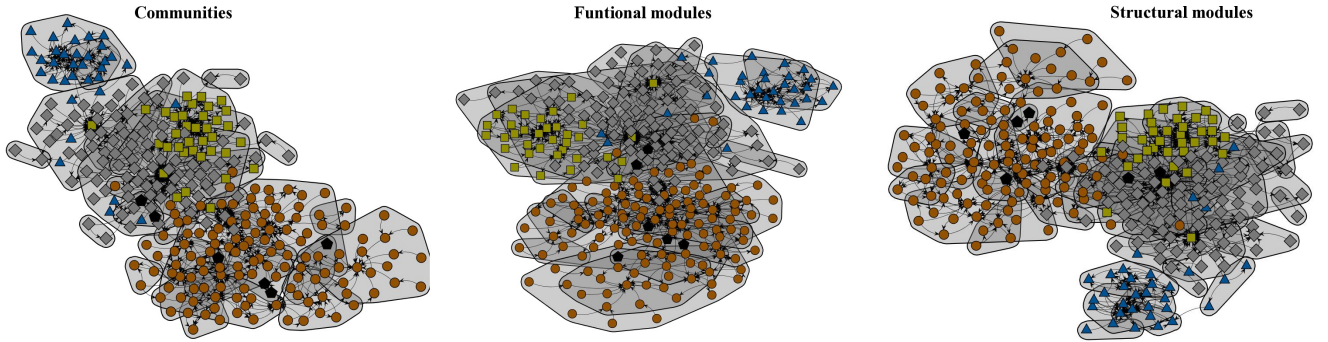


Figure 6: (left) Communities representing highly modular structure of the software system [42, 41]. (middle) Functional modules that represent highly functional partitioning of the system [44, 43]. (right) General structural modules conveying modular and functional links (bottom-most level of the hierarchy in Figure 5).

Table 5: Normalized mutual information [10] (NMI) between software packages and identified network modules, $NMI \in [0, 1]$. Number of modules is shown in small font.

Network		MO		CP		MM		GP	
<i>flmng</i>	16	0.580	14	0.609	27	0.521	16	0.610	26
<i>colt</i>	19	0.519	10	0.473	20	0.533	19	0.530	26
<i>jung</i>	39	0.614	13	0.650	30	0.661	39	0.680	41
<i>org</i>	47	0.503	11	0.537	30	0.378	39	0.536	33
<i>weka</i>	81	0.558	26	0.410	49	0.430	63	0.314	28
<i>javax</i>	107	0.704	59	0.761	155	0.392	89	0.747	192

Table 6: Classification accuracy (CA) for software package prediction, $CA \in [0, 1]$. (l_∞ is the number of levels of the package hierarchy, whereas l is the average level for a software class. Value under P_i corresponds to CA for the i -th level of the hierarchy.)

Network	l	l_∞	P	P_4	P_3	P_2	P_1
<i>flmng</i>	2.65	4	0.566	\leftarrow	0.572	0.793	1.000
<i>colt</i>	3.35	4	0.654	\leftarrow	0.756	0.942	1.000
<i>jung</i>	2.97	4	0.617	\leftarrow	0.663	0.857	1.000
<i>org</i>	3.50	7	0.616	0.616	0.714	0.989	1.000
<i>weka</i>	3.02	6	0.684	0.692	0.736	0.871	1.000
<i>javax</i>	3.11	5	0.626	0.631	0.816	0.982	1.000

Table 7: Software project quality indicators presented in the study. For each indicator, we give the range and the expected value of a well designed software system (based on Section 3).

Quality indicator	Expected value	Range	Comment
p_k^{in}	$\sim k^{-\gamma_{in}}$	$\gamma_{in} > 1$	High code reusability (Section 3.1).
k^{in}	$\gg 0$	∞	
p_k^{out}	$\sim k^{-\gamma_{out}}$	$\gamma_{out} > 1$	Low code complexity (Section 3.1).
k^{out}	$\ll n$	∞	
D	$\gg 0, \ll 1$	$[0, 1]$	Characteristics of the project domain [43].
$l - l_{ER}$	≤ 0	∞	Well structured and designed project (Section 3.2).
E	≈ 0	$[0, 1]$	Low efficiency of information flow (Section 3.2).
n_d/n	$\gg 0$	$[0, 1]$	Low project controllability (Section 3.3).
γ	$\ll 3$	> 1	Low project vulnerability, high robustness (Section 3.1).

Table 8: Software class indicators presented in the study. For each indicator, we give the range and the expected value of highly influential, most vulnerable or high complexity classes (based on Section 3).

Class indicator	Expected value	Range	Comment
DC, BC	$\gg 0$	$[0, 1]$	Highly influential seed classes (Section 3.3).
CC			Highly vulnerable seed classes (Section 3.3).
k^{in}	$\gg 0$	∞	Highly influential hub classes (Section 3.1).
k^{out}			High complexity hub classes (Section 3.1).

ity indicators identified in the study. Indicators can be employed to assess project structure and design, code complexity and reusability, controllability and vulnerability, information flow, and other. Due to space limitations, comparison with other approaches for measuring software quality is omitted (e.g., metrics of coupling and cohesion [34]).

5. CONCLUSIONS

The paper conducts a comprehensive study of software networks constructed from Java source code. First, we address macroscopic network properties that are related to structural design of the corresponding software project. Next, we analyze the networks on a microscopic level of nodes, to highlight most influential and vulnerable software classes. Last, we analyze mesoscopic network structural modules and expose their applicability in project refactoring. Among other, we show that software systems are highly vulnerable to processes like bug propagation, however, they are not easily controllable. On the other hand, Java language can be controlled through merely 17% of `java` namespace. We also identify several network-based quality indicators that can be employed to assess software project design, reusability, robustness, controllability and other. The study thus exposes network analysis as a prominent set of tools for software systems engineering.

6. ACKNOWLEDGMENTS

This work has been supported by the Slovene Research Agency ARRS within Research Program No. P2-0359.

7. REFERENCES

- [1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, algorithms, and applications*. Prentice-Hall, Upper Saddle River, NJ, 1993.
- [2] R. Albert, H. Jeong, and A. L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, 2000.
- [3] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [4] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 397–412, 2006.
- [5] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
- [6] K. Cai and B. Yin. Software execution processes as an evolving complex network. *Information Sciences*, 179(12):1903–1928, 2009.
- [7] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2000.
- [8] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, 2004.
- [9] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.
- [10] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, P09008, 2005.
- [11] M. Egerstedt. Complex networks: Degrees of control. *Nature*, 473(7346):158–159, 2011.
- [12] P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.
- [13] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
- [14] L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1979.
- [15] J. Gao, S. V. Buldyrev, H. E. Stanley, and S. Havlin.

Networks formed from interdependent networks. *Nature Physics*, 8(1):40–48, 2012.

- [16] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of United States of America*, 99(12):7821–7826, 2002.
- [17] D. Hyland-Wood, D. Carrington, and S. Kaplan. Scale-free nature of java software package, class and method collaboration graphs. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 1–10, 2006.
- [18] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.
- [19] G. A. Kohring. Complex dependencies in large software systems. *Advances in Complex Systems*, 12(6):565–581, 2009.
- [20] A. Kuhn, S. Ducasse, and T. Gırba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.
- [21] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *e-print arXiv:cs/0411096v1*, 2004.
- [22] V. Latora and M. Marchiori. Efficient behavior of small-world networks. *Physical Review Letters*, 87(19):198701, 2001.
- [23] Y. Liu, J. Slotine, and A. Barabasi. Controllability of complex networks. *Nature*, 473(7346):167–173, 2011.
- [24] A. Lombardi and M. Hörnquist. Controllability analysis of networks. *Physical Review E*, 75(5):056110, 2007.
- [25] S. Milgram. The small world problem. *Psychology Today*, 1(1):60–67, 1967.
- [26] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, and J. Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 328(5980):876–878, 2010.
- [27] C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(2), 2003.
- [28] G. M. Narayan, K. Gopinath, and V. Sridhar. Structure and interpretation of computer programs. In *Proceedings of the IEEE International Symposium on Theoretical Aspects of Software Engineering*, 2008.
- [29] M. E. J. Newman and E. A. Leicht. Mixture models and exploratory analysis in networks. *Proceedings of the National Academy of Sciences of United States of America*, 104(23):9564, 2007.
- [30] R. Pastor-Satorras and A. Vespignani. Epidemic spreading in scale-free networks. *Physical Review Letters*, 86(14):3200–3203, 2001.
- [31] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Communications of the ACM*, 48(5):99–103, 2005.
- [32] S. Sinha. Few and far between. *Physics*, 4:81, 2011.
- [33] S. N. Soffer and A. Vázquez. Network clustering coefficient without degree-correlation biases. *Physical Review E*, 71(5):057101, 2005.
- [34] W. P. Stevens, G. J. Myers, and L. L. Constantive. Structured design. *IBM Systems Journal*, 38(2):231–256, 1999.
- [35] S. Valverde, R. F. Cancho, and R. V. Solé. Scale-free networks from optimal design. *Europhysics Letters*, 60(4):512, 2002.
- [36] S. Valverde and R. V. Solé. Logarithmic growth dynamics in software networks. *Europhysics Letters*, 72(5):858–864, 2005.
- [37] S. Valverde and R. V. Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2):026107, 2005.
- [38] S. Valverde and R. V. Solé. Hierarchical small worlds in software architecture. *Dynamics of Continuous, Discrete and Impulsive Systems - Series B*, 14:1–11, 2007.
- [39] L. Šubelj and M. Bajec. Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications*, 390(16):2968–2975, 2011.
- [40] L. Šubelj and M. Bajec. Generalized network community detection. In *Proceedings of the ECML PKDD Workshop on Finding Patterns of Human Behaviors in Network and Mobility Data*, pages 66–84, Athens, Greece, 2011.
- [41] L. Šubelj and M. Bajec. Robust network community detection using balanced propagation. *European Physical Journal B*, 81(3):353–362, 2011.
- [42] L. Šubelj and M. Bajec. Unfolding communities in large complex networks: Combining defensive and offensive label propagation for core extraction. *Physical Review E*, 83(3):036103, 2011.
- [43] L. Šubelj and M. Bajec. Clustering assortativity, communities and functional modules in real-world networks. *e-print arXiv:12023188v1*, 2012.
- [44] L. Šubelj and M. Bajec. Ubiquitousness of link-density and link-pattern communities in real-world networks. *European Physical Journal B*, 85(1):32, 2012.
- [45] J. Wang and Y. Liu. Modeling software faults propagation. *Europhysics Letters*, 92(6):60009, 2010.
- [46] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.
- [47] L. Wen, D. Kirk, and R. G. Dromey. Software systems as complex networks. In *Proceedings of the IEEE International Conference on Cognitive Informatics*, pages 106–115, 2007.