Predicting Defects using Network Analysis on Dependency Graphs

Thomas Zimmermann[+]
University of Calgary
Calgary, Alberta, Canada
tz@acm.org

Nachiappan Nagappan
Microsoft Research
Redmond, Washington, USA
nachin@microsoft.com

ABSTRACT
In software development, resources for quality assurance are limited by time and by cost. In order to allocate resources effectively, managers need to rely on their experience backed by code complexity metrics. But often dependencies exist between various pieces of code over which managers may have little knowledge. These dependencies can be construed as a low level graph of the entire system. In this paper, we propose to use network analysis on these dependency graphs. This allows managers to identify central program units that are more likely to face defects. In our evaluation on Windows Server 2003, we found that the recall for models built from network measures is by 10% points higher than for models built from complexity metrics. In addition, network measures could identify 60% of the binaries that the Windows developers considered as critical—twice as many as identified by complexity metrics.

Categories and Subject Descriptors
D.2.8 **[Software Engineering]:** Metrics—*Performance measures, Process metrics, Product metrics.* D.2.9 **[Software Engineering]:** Management—*Software quality assurance (SQA)*

General Terms
Management, Measurement, Reliability, Experimentation

1. INTRODUCTION
Software errors cost the U.S. industry 60 billion dollars a year according to a study conducted by the National Institute of Standards and Technology [48]. One contributing factor to the high number of errors is the limitation of resources for quality assurance (QA). Such resources are always limited by *time*, e.g., the deadlines that development teams face, and by *cost*, e.g., not enough people are available for QA. When managers want to spend resources most effectively, they would typically allocate them on the parts where they expect most defects or at least the most severe ones. Put in other words: *based on their experience, managers predict the quality* of the product to make further decisions on testing, inspections, etc.

In order to support managers with this task, research identified several quality indicators and developed prediction models to predict the quality of software parts. The complexity of source code is one of the most prominent indicators for such models. However, even though several studies showed McCabe's cyclomatic complexity to correlate with the number of defects [2, 35, 47], there is no universal metric or prediction model that applies to all projects [35]. One drawback of most complexity metrics is that they only focus on single elements, but rarely take the interactions between elements into account. However, with the advent of static and dynamic bug localization techniques, the nature of defects has changed and today most defects in bug databases are of semantic nature [26].

In this paper we will pay special attention to interactions between elements. More precisely, we will investigate how dependencies correlate with and predict defects for binaries in Windows Server 2003. While this is not the first work on defects and dependencies, we will cover a different angle: In order to identify the binaries that are most central in Windows Server 2003, we apply *network analysis* on dependency graphs. Network analysis is very popular in social sciences which studies networks between humans (actors) and their interactions (ties). In our context the binaries are the "actors" and the dependencies are the "ties".

The outline of this paper is as follows: We will motivate our study with two special dependency structures. For *central binaries* (in terms of network analysis) we could observe a substantial correlation with defects (Section 2). After a discussion of related work (Section 3), we will present the data collection for our study: for Windows Server 2003 we computed dependencies, complexity metrics, and measures from network analysis (Section 4). In our experiments, we will evaluate network measures against complexity metrics. Additionally, we show that network analysis succeeds in identifying binaries that are considered as most harmful by developers (Section 5). We close our paper with conclusion and consequences (Section 6).

2. MOTIVATION
When we analyzed defect data and dependency graphs for Windows Server 2003, we made the following observations.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

[+] Tom Zimmermann was an intern with the Software Reliability Research Group, Microsoft Research in the summer of 2006 when this work was carried out.

Central binaries tend to be defect-prone. We identified several network motifs in the dependency graph of Windows Server 2003. Network motifs are patterns that describe similar, but not necessarily isomorphic subgraphs; originally they were introduced in biological research [29]. One of the motifs for Windows Server 2003 looks like a star (see Figure 1): it consists of a binary B that is connected to the main component of the dependency graph. Several other "satellite" binaries surround B and exclusively depend on binary B. In most occurrences of the pattern, the binary B was defect-prone, while the satellite binaries were defect-free. Network analysis identifies binary B as *central* (a so-called 'Broker") in the dependency graph because it controls its satellite binaries. We conjecture that binaries that are identified as central by network analysis are more defect-prone than others.

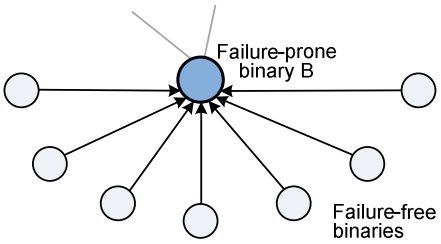


Figure 1. Star pattern.

The larger a clique, the more defect-prone are its binaries. A clique is a set of binaries for which between every pair of binaries (X, Y) a dependency exists—we neglect the direction, i.e., it does not matter whether X depends on Y, Y on X, or both. Figure 2 shows an example for an undirected clique; a clique is maximal if no other binary can be added without losing the clique property. We enumerated all *maximal undirected cliques* in the dependency graph of Windows Server 2003 with the Bron-Kerbosch algorithm [12]. The enumeration of cliques is a core component in many biological applications. Next we grouped the cliques by size and computed the *average number of defects* per binary. Figure 3 shows the results, including a 95% confidence interval of the average. We can observe that the *average number of defects increases with the size of the clique a binary resides in*. Put in another way, binaries that are part of more complex areas (cliques) have more defects.

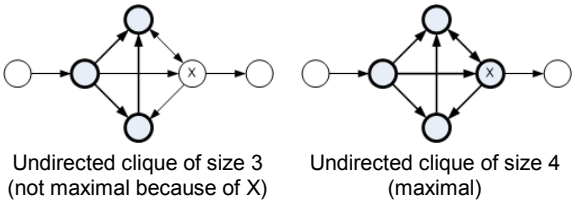


Figure 2. Undirected cliques.

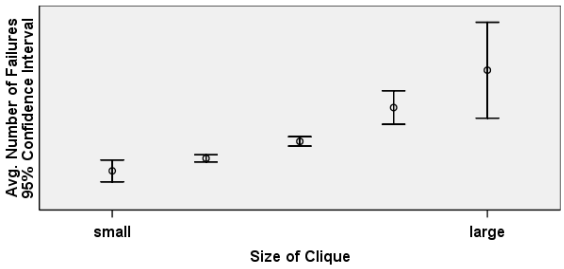


Figure 3. Defect-proneness of binaries in cliques.

Again, this observation motivates network analysis: binaries that are part of cliques are close to each other, which is measured by the network measure *closeness*. We hypothesize that closeness, as well as other network measures, correlates with the number of defects.

In this paper, we will compute measures from network analysis on dependency graphs. More formally, the hypotheses that we will investigate are the following:

H1	*Network measures on dependency graphs can indicate critical binaries that are missed by complexity metrics.*
H2	*Network measures on dependency graphs correlate positively with the number of post-release defects*—an increase in a measure is accompanied by an increase in defects.
H3	*Network measures on dependency graphs, can predict the number of post-release defects.*

3. RELATED WORK

In this section we discuss related work; it falls into three categories: social network analysis in software engineering, software dependencies, and complexity metrics.

3.1 NETWORK ANALYSIS IN SE

The use of network analysis is not new to software engineering. Several researchers used social network analysis to study the dynamics of open source development. Ghosh showed that many SourceForge.net projects are organized as self-organizing social networks [18]. Madley et al. conducted a similar study where they focused on collaboration aspects by looking at the joint-membership of developers in projects [28]. In addition to committer networks, Lopez et al. investigated module networks that show how several modules relate to each other [27]. Ohira et al. used social networks and collaborative filtering to support the identification of experts across projects [38]. Huang et al. used historical data to identify core and peripheral development teams in software projects [22].

Social network analysis was also used on research networks. Hassan and Holt analyzed the reverse engineering community using co-authorship relations. They also identified emerging research trends and directions over time and compared reverse engineering to the entire software engineering community [20].

In contrast to these approaches, we do not analyze the relations between developers or projects, but rather between binaries of a single project. Also the objective of our study is different. While most of the existing work considered organizational aspects, our aim is to predict defects.

3.2 SOFTWARE DEPENDENCIES

Pogdurski and Clarke [42] presented a formal model of program dependencies as the relationship between two pieces of code inferred from the program text. Program dependencies have also been analyzed in terms of testing [25], code optimization and parallelization [17], and debugging [40]. Empirical studies have also investigated dependencies and program predicates [7] and inter-procedural control dependencies [45] in programming language research. Bevan and Whitehead combined dependency graphs and historic data to identify software instabilities [4, 5].

The information-flow metric defined by Henry and Kafura [21], uses *fan-in* (a count of the number of modules that call a given module) and *fan-out* (a count of the number of modules that are called by a given module) to calculate a complexity metric. Components with a large fan-in and large fan-out may indicate poor design. In contrast, our work uses not only calls, but also data dependencies. Furthermore, we distinguish between different types of dependencies such as intra-dependencies and outgoing dependencies.

Schröter et al. [44] showed that the actual import dependencies (not just the count) can predict defects, e.g., importing compiler packages is riskier than importing UI packages. Earlier work on dependencies at Microsoft [33] showed that code churn and dependencies can be used as efficient indicators of post-release defects. The basic idea being, for example suppose that component A has many dependencies on component B. If the code of component B changes (churns) a lot between versions, we may expect that component A will need to undergo a certain amount of churn in order to keep in synch with component B. That is, churn often will propagate across dependencies. Together, a high degree of dependence plus churn can cause errors that will propagate through a system, reducing its reliability. This work was extended to predict defects for subsystem by taking the complexity of dependency graphs into account [50].

3.3 COMPLEXITY METRICS

Typically, research on defect-proneness captures software complexity with metrics and builds models that relate these metrics to failure-proneness [34]. Basili et al. [2] were among the first to validate that OO metrics predict defect density. Subramanyam and Krishnan [47] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects. Briand et al. [9] identified several coupling measures for C++ that could serve as early quality indicators for the design of a project.

Our experiments focus on post-release defects since they matter most for the end-users of a program. Only few studies addressed post-release defects: Binkley and Schach [6] developed a coupling metric and showed that it outperforms several other metrics; Ohlsson and Alberg [39] used metrics to predict modules that fail during operation. Additionally, within five Microsoft projects, Nagappan et al. [35] identified metrics that predict post-release defects and reported how to systematically build predictors for post-release defects from history. In contrast to their work, we develop new metrics on dependency data from a graph theoretic point of view.

4. DATA COLLECTION

For our experiments we build a dependency graph of Windows Server 2003 (Section 4.1) and we compute network measures on it (Section 4.2). Additionally, we collect complexity metrics (Section 4.3) which we use to quantify the contribution of network analysis.

4.1 DEPENDENCY GRAPH

A software dependency is a directed relation between two pieces of code (such as expressions or methods). There exist different kinds of dependencies: *data dependencies* between the definition and use of values and *call dependencies* between the declaration of functions and the sites where they are called. Microsoft has an automated tool called MaX [46] that tracks dependency information at the function level, including calls, imports, exports, RPC, COM, and Registry access. MaX generates a system-wide dependency graph from both native x86 and .NET managed binaries. Within Microsoft, MaX is used for change impact analysis and for integration testing [46].

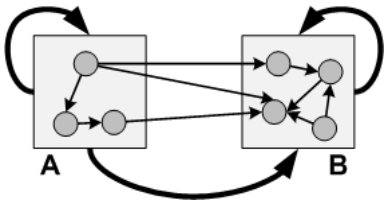


Figure 4. Lifting up dependencies

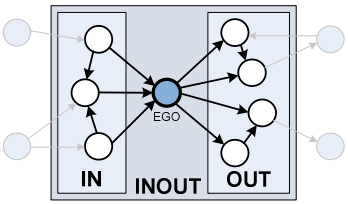


Figure 5. Different neighborhoods

For our analysis, we generated a system-wide dependency graph with MaX on function level. Since binaries are the lowest level of granularity to which defects can be accurately mapped back to, we lifted this graph up to binary level in a separate post-processing step (that is sketched in Figure 4). In this paper, we consider only the *presence* of dependencies such as A depends on B, i.e., we neglect the multiplicity of dependencies such as A depends three times on B.

For our experiments, we define a dependency graph as a directed graph G = (V, E) where V is a set of nodes (=binaries) and $E \subseteq V \times V$ is a set of edges (=dependencies). Note that we allow self-edges, i.e., a binary can depend on itself.

4.2 NETWORK MEASURES

On the dependency graph we computed for each node (binary) a number of network measures by using the Ucinet 6 tool [8]. In this section, we will describe these measures more in detail, however, for or a more comprehensive overview, we refer to textbooks on network analysis [19, 37, 49].

EGO NETWORKS VS. GLOBAL NETWORKS

One important distinction made in network analysis is between *ego networks* and *global networks*.

Every node in a network has a corresponding ego network that describes how the node is connected to its neighbors. (Nodes are often referred to as "ego" in network analysis.) Figure 5 explains how ego networks are constructed. In our case, they contain the ego binary itself, binaries that depend on the ego (IN), binaries on which the ego depends (OUT), and the dependencies between these binaries. The ego network would thus be the subgraph within the INOUT box of Figure 5.

Table 1. Network measures for ego networks

Measure	Description
Size	The size of the ego network is the number of nodes.
Ties	The number of directed ties corresponds to the number of edges.
Pairs	The number of ordered pairs is the maximal number of directed ties, i.e., *Size×(Size–1)*.
Density	The percentage of possible ties that are actually present, i.e., *Ties/Pairs*.
WeakComp	The number of weak components (=sets of connected binaries) in neighborhood.
nWeakComp	The number of weak components normalized by size, i.e., *WeakComp/Size*.
TwoStepReach	The percentage of nodes that are two steps away.
ReachEfficency	The reach efficiency normalizes TwoStepReach by size, i.e., *TwoStepReach/Size*. High reach efficiency indicates that ego's primary contacts are influential in the network.
Brokerage	The number of pairs not directly connected. The higher this number, the more paths go through ego, i.e., ego acts as a "broker" in its network.
nBrokerage	The Brokerage normalized by the number of pairs, i.e., *Brokerage/Pairs*.
EgoBetween	The percentage of *shortest* paths between neighbors that pass through ego.
nEgoBetween	The Betweenness normalized by the size of the ego network.

In contrast, the global network corresponds always to the entire dependency graph. While ego networks allow us to measure the local importance of a binary with respect to its neighbors, global networks reveal the importance of a binary within the entire software system. Since we expected local and global importance to complement each other, we used both in our study.

EGO NETWORKS

An ego network for a binary consists of its neighborhood in the dependency graph. We distinguish between three kinds of neighborhoods (see also Figure 5):

- *In-neighborhood (IN)* contains the binaries that depend on the ego binary.
- *Out-neighborhood (OUT)* contains the binaries on which the ego binary depends.
- *InOut-neighborhood (INOUT)* is the combination of the In- and Out-neighborhood.

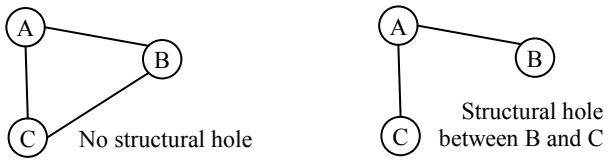
For every binary, we induce its three ego networks (one for each kind of neighborhood) and compute fairly basic measures that are listed in Table 1. Additionally, we compute measures for structural holes that are described below.

GLOBAL NETWORK

Within the global network (=dependency graph) we can measure the importance of binaries for the whole software system and not only their local neighborhood. For most network measures we use directed edges; however, some measures can be applied to symmetric, undirected networks *(Sym)* or ingoing *(In)* and outgoing *(Out)* edges respectively. On the global network, we compute measures for structural holes and centrality. Both concepts are summarized below.

STRUCTURAL HOLES

The term of structural holes was coined by Ronald Burt [13]. Ideally, the influence of actors is balanced in social networks. The Figure below shows two networks for three actors A, B, and C.



In the left network all actors are tied to each other and therefore have the same influence. In the network on the right hand side, the tie between B and C is missing ("structural hole"), giving A an advanced position over B and C.

We used the following measures related to structural holes in our study of dependency graphs:

- *Effective size of network (EffSize)* is the number of binaries that are connected to a binary X minus the average number of ties between these binaries. Suppose X has three neighbors that are not connected to each other, then the effective size of X's ego network is 3–0=3. If each of the three neighbors would be connected to the other ones, the average number of ties would be two, and the effective size of X's ego network reduces to 3–2=1.
- *Efficiency* norms the effective size of a network to the total size of the network.
- *Constraint* measures how strongly a binary is constrained by its neighbors. The idea is that neighbors that are connected to other neighbors can constrain a binary. For more details we refer to Burt [13].
- *Hierarchy* measures how the constraint measure is distributed across neighbors. When most of the constraint comes from a single neighbor, the value for hierarchy is higher. For more details we refer to Burt [13].

The values for the above measures are higher for binaries with neighbors that are closely connected to each other and other binaries. One might expect that such complex dependency structures result in a higher number of defects.

CENTRALITY MEASURES

One of the most frequently used concepts in network analysis is *centrality* [43]. It is used to identify actors that are in "favored positions". Applied on dependency graphs, centrality identifies the binaries that are specially exposed to dependencies, e.g., by being the target of many dependents. There are different approaches to measure centrality:

- *Degree centrality*. The degree measures the number of dependencies for a binary. The idea for dependency graphs is that binaries with many dependencies are more defect-prone than others.

Table 2. Metrics used in our Windows study

Metric	Description
Module metrics for a binary B:	
Function	# functions in B
GlobalVariables	# global variables in B
Per-function metrics for a function f():	
Lines	# executable lines in f()
Parameters	# parameters in f()
FanIn	# functions calling f()
FanOut	# functions called by f()
Complexity	McCabe's cyclomatic complexity of f()
OO metrics for a class C	
ClassMethods	# methods in C
SubClasses	# subclasses of C
InheritanceDepth	Depth of C in the inheritance tree
ClassCoupling	Coupling between classes
CyclicClassCoupling	Cyclic coupling between classes

Table 3. Recall for Escrow binaries

Network measures	Recall
GlobalInClosenessFreeman	0.60
GlobalIndwReach	0.60
EgoInSize	0.55
EgoInPairs	0.55
EgoInBroker	0.55
EgoInTies	0.50
GlobalInDegree	0.50
GlobalBetweenness	0.50
…	…
Complexity metric	**Recall**
TotalParameters	0.30
TotalComplexity	0.30
TotalLines	0.30
TotalFanIn	0.30
TotalFanOut	0.30
…	….

- *Closeness centrality*. While degree centrality measures only the immediate dependencies of a binary, closeness centrality additionally takes the distance to all other binaries into account. There are different variants to compute closeness:
  - *Closeness* is the sum of the lengths of the shortest (geodesic) paths from a binary (or to a binary) from all other binaries. There exist different variations of closeness in network analysis. Our definition corresponds to the one used by Freeman (see [19, 37, 49]).
  - *dwReach* is the number of binaries that can be reached from a binary (or which can reach a binary). The distance is weighted by the number of steps with factors 1/1, 1/2, 1/3, etc.
  - *Eigenvector centrality* is similar to Google's PageRank value [14]; it assigns relative scores to all binaries in the dependency graphs. Dependencies to binaries having a high score contribute more to the score of the binary in question.
  - *Information centrality* is the harmonic mean of the length of paths ending at a binary. The value is smaller for binaries that are connected to other binaries through many short paths.

  Again, the hypothesis is that the more central a binary is, the more defects it will have,
- *Betweenness centrality* measures for a binary on how many shortest paths between other binaries it occurs. The hypothesis is that binaries that are part of many shortest paths are more likely to contain defects because defects propagate.

4.3 COMPLEXITY METRICS

In order to quantify the contribution of network analysis on dependency graphs, we use code metrics as a control set for providing a comparison point. For each binary, we computed several code metrics, described in Table 2. These metrics apply to a binary *B* and to a function or method *f()*, respectively. In order to have all metrics apply to binaries, we summarized the function metrics across each binary. For each function metric *X*, we computed the *total* and the *maximum* value per binary (denoted as *TotalX* and *MaxX*, respectively). As an example, consider the *Lines* metric, counting the number of executable lines per function. The *MaxLines* metric indicates the length of the largest function in B, while *TotalLines*, the sum of all *Lines*, represents the total number of executable lines in B.

5. EXPERIMENTAL ANALYSIS

In this section, we will support our hypotheses that network analysis of dependency graphs helps to predict the number of defects for binaries.

We carried out several experiments for Windows Server 2003: First we show that network analysis can identify critical "escrow" binaries (Section 5.1). We continue with a correlation analysis of network measures, metrics, and number of defects (Section 5.2) and regression models for defects prediction (Section 5.3). Finally, we present threats to validity (Section 5.4).

5.1 ESCROW ANALYSIS

The development teams of Windows Server 2003 maintain a list of critical binaries that are called *escrow binaries*. Whenever programmers change an escrow binary, they must adhere to a special protocol to ensure the stability of Windows Server. This protocol involves more extensive testing, fault-inject, code reviews etc. on the binary and its related dependencies. In other words these escrow binaries are the "most important" binaries in Windows. An example escrow binary is the Windows kernel binary. The developers manually select the binaries in the escrow based on past experience with previous builds, changes, and defects.

We used the network measures and complexity metrics (from Sections 4.2 and 4.3) to predict the list of escrow binaries. For each measure/metric, we ranked the binaries according to its value and took the top N binaries as the prediction, with N being the size of the escrow list. In order to evaluate the predictions, we computed the recall that is the percentage of escrow binaries that we successfully could retrieve. In order to protect proprietary information, i.e., the size of the escrow list, we report only percentages that are truncated to the next multiple of 5%. For instance, the percentage of 23% would be reported as 20%.

The results in Table 3 show that complexity metrics fail to predict escrow binaries. They can retrieve only 30%, while the network measures for closeness centrality *can retrieve twice as much*. This

observation supports our first hypothesis that *network measures on dependency graphs can indicate critical binaries that are missed by complexity metrics (H1)*. Being complex does not make a binary critical in software development—it is more likely the combination of being complex and central to the system.

5.2 CORRELATION ANALYSIS

In order to investigate our hypothesis H2, we determined the Pearson and Spearman rank correlation between the number of defects and each network measure (Section 4.2) as well as each complexity metric (Section 4.3). The Pearson bivariate correlation requires data to be distributed normally and the association between elements to be linear. In contrast, the Spearman rank correlation is a robust technique that can be applied even when the association between values is non-linear [16]. For completeness we compute both correlations coefficients. The closer the value of correlation is to –1 or +1, the higher two measures are correlated—positively for +1 and negatively for –1. A value of 0 indicates that two measures are independent.

The Spearman correlation values for Windows Server 2003 are shown in Table 4. The table consists of three parts: ego network measures, global network measures, and complexity metrics. The columns distinguish between different neighborhoods (IN, OUT, INOUT) and directions of edges (ingoing, outgoing, symmetric). Correlations that are significant at 0.99 are indicated with (*). The values for Pearson correlation are listed in a similar table in the appendix (Table 5). We can make the following observations.

(1) Some network measures do not correlate with the number of defects. The correlations for the number of weak components in a neighborhood (WeakComp), the Hierarchy and the Efficiency are all close to zero, which means that their values and the number of defects are independent.

(2) Some network measures have negative correlation coefficients. The normalized number of weak components in a neighborhood (nWeakComp) as well as the Reach Efficiency and the Constraint show a negative correlation between –0.424 and –0.463. This means that an increase in centrality comes with a decrease in number of defects. Since the values for the aforementioned measures are higher for binaries with neighbors that are closely connected to each other and other binaries, this suggests that being in a closely connected neighborhood does not necessarily result in a high number of defects. This explanation is also supported by the negative correlation of –0.320 for Density.

(3) Network measures have higher correlations for OUT and INOUT than for IN neighborhoods. In other words, outgoing dependencies are more related to defects than ingoing dependencies. Schröter et al. found similar evidence and used the targets of outgoing dependencies to predict defects [44]. The measures with the highest observed correlations were related to the *size of the neighborhoods* (Size, Pairs, Broker, EffSize, and Degree) and to *centrality* (Eigenvector and Information), all of them had correlations of 0.400 or higher.

(4) Most complexity metrics have slightly higher correlations than network measures. For non-OO metrics the correlations are above 0.500. In contrast, for OO metrics the correlations are lower (around 0.300) because not all parts of Windows Server 2003 are developed with object-oriented programming languages. This shows that OO metrics are only of limited use for predicting defects in heterogeneous systems.

Table 4. Spearman correlation between the number of defects and network measures as well as complexity metrics. Correlations significant at 99% are marked by (**). Correlations above 0.40 are printed in boldface.

	Spearman Correlation		
Ego Network	**In**	**Out**	**InOut**
Size	.283(**)	**.440(**)**	**.462(**)**
Ties	.245(**)	**.434(**)**	**.455(**)**
Pairs	.276(**)	**.440(**)**	**.462(**)**
Density	.253(**)	-.273(**)	-.320(**)
WeakComp	.274(**)	.035	.082(**)
nWeakComp	.227(**)	**-.438(**)**	**-.453(**)**
TwoStepReach	.287(**)	.326(**)	.333(**)
ReachEfficency	.230(**)	**-.402(**)**	**-.424(**)**
Brokerage	.271(**)	**.438(**)**	**.461(**)**
nBrokerage	.283(**)	.275(**)	.321(**)
EgoBetween	.263(**)	.292(**)	.320(**)
nEgoBetween	.279(**)	.294(**)	.285(**)
EffSize			**.466(**)**
Efficiency			.262(**)
Constraint			**-.463(**)**
Hierarchy			.064(**)
Global Network			
Eigenvector			**.428(**)**
Fragmentation			.276(**)
Betweenness			.319(**)
Information			**.446(**)**
Power			.397(**)
EffSize			**.455(**)**
Efficiency			.021
Constraint			**-.454(**)**
Hierarchy			.176(**)
	Ingoing	**Outgoing**	**Symmetric**
Closeness	-.057(**)	.284(**)	.372(**)
Degree	.283(**)	**.440(**)**	**.462(**)**
dwReach	.285(**)	.394(**)	.379(**)
Complexity Metrics		**Max**	**Total**
Functions			**.507(**)**
GlobalVariables			**.436(**)**
Lines		.317(**)	**.516(**)**
Parameters		.386(**)	**.521(**)**
FanIn		**.452(**)**	**.502(**)**
FanOut		.360(**)	**.493(**)**
Complexity		.310(**)	**.509(**)**
OO Metrics		**Max**	**Total**
ClassMethods		.315(**)	.336(**)
SubClasses		.296(**)	.295(**)
InheritanceDepth		.286(**)	.308(**)
ClassCoupling		.318(**)	.327(**)
CyclicClassCoupling			.331(**)

To summarize, we could observe significant correlations for most network measures, and most of them were positive and moderate. However, since we observed several negative correlations, we need to remove the "positively" from our initial hypothesis (H2). The revised hypothesis that *network measures on dependency graphs correlate with the number of post-release defects (H2*)* is confirmed by our observations. At a first glance complexity metrics might outperform network measures, but we show in Section 5.3 that network measures improve prediction models for defects.

5.3 REGRESSION ANALYSIS

Since network measures on dependency graphs correlate with post-release defects, can we use them to predict defects? To answer this question, we build multiple linear regression (MLR) models where the number of post-release defects forms the dependant variable. We build separate models for three different sets of input variables:

SNA. This set of variables consists of the network measures that were introduced in Section 4.2.

METRICS. This set consists of all non-OO complexity metrics listed in Table TAB. We decided to ignore OO-metrics for the regression analysis because they were only applicable to a part of Windows Server 2003 because most of Windows is comprised of non-OO code.

SNA+METRICS. This set is the combination of the two previous sets (SNA, METRICS) and allows us to quantify the value added by network measures.

We carried out six experiments: one for each combination out of two kinds of regression models (linear, logistic) and three sets of input variables (SNA, METRICS, SNA+METRICS).

PRINCIPAL COMPONENT ANALYSIS

One difficulty associated with MLR is *multicollinearity* among the independent variables. Multicollinearity comes from inter-correlations amongst metrics such as between the aforementioned *Multi_Edges* and *Multi_Complexity*. Inter-correlations can lead to an inflated variance in the estimation of the dependant variable. To overcome this problem, we use a standard statistical approach called *Principal Component Analysis* (PCA) [23].

With PCA, a small number of uncorrelated linear combinations of variables are selected for use in regression (linear or logistic). These combinations are independent and thus do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible—for our experiments we selected principal components that account for a cumulative sample variance greater than 95%.

We ended up with 15 principal components for SNA, 6 for METRICS, and 20 for the SNA+METRICS set of measures. The principal components were then used as the independent variables in the linear and logistic regression models.

TRAINING REGRESSION MODELS

To evaluate the predictive power of graph complexities we use a standard evaluation technique: *data splitting* [31]. That is, we randomly pick two-thirds of all binaries to build a prediction model and use the remaining one-third to measure the efficacy of the built model (see Figure 6). For every experiment, we performed 50 random splits to ensure the stability and repeatability of our results—in total we trained 300 models. Whenever possible, we reused the random splits to facilitate comparison of results.

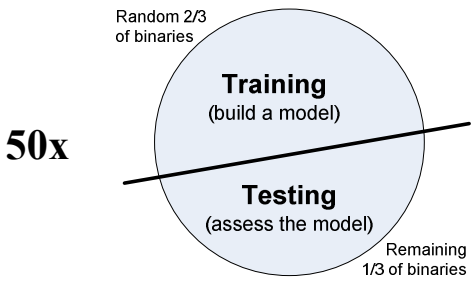


Figure 6: Random split experiments

We measured the quality of *trained* models with:

- The **R^2 value** is the ratio of the regression sum of squares to the total sum of squares. It takes values between 0 and 1, with larger values indicating more variability explained by the model and less unexplained variation—a high R^2 value indicates good explanative power, but *not* predictive power. For logistic regression models, a specialized R^2 value introduced by Nagelkerke [36] is typically used.
- The **adjusted R^2 measure** also can be used to evaluate how well a model fits a given data set [1]. It explains for any bias in the R^2 measure by taking into account the degrees of freedom of the independent variables and the sample population. The adjusted R^2 tends to remain constant as the R^2 measure for large population samples.

Additionally, we performed **F-tests** on the regression models. Such tests measure the statistical significance of a model based on the null hypothesis that its regression coefficients are zero. In our case, every model was significant at 99%.

LINEAR REGRESSION

In order to test how well linear regression models predict defects, we computed the Pearson and Spearman correlation coefficients (see Section 5.2) between the predicted number of defects and the actual number of defects. As before, the closer a value to –1 or +1, the higher two measures are correlated—in our case values close to 1 are desirable. In Figures 7 and 8, we report only correlations that were significant at 99%.

Figure 7 shows the results of the three experiments (SNA, METRICS, and SNA+METRICS) for linear regression modeling, each of them consisting of 50 random splits. For all three experiments, we observe consistent R^2 and adjusted R^2 values. This indicates the efficacy of the models built using the random split technique. The values for Pearson are less consistent; still we can observe high correlations (above 0.60).

The values for Spearman correlation values indicate the sensitivity of the predications to estimate defects—i.e., an increase/decrease in the estimated values is accompanied by a corresponding increase/decrease in the actual number of defects. In all three experiments (SNA, METRICS, SNA+METRICS), the values for Spearman correlation are consistent across the 50 random splits. For SNA and METRICS separately the correlations are close to 0.50. This means that models built from network measures can predict defects as well as models built from complexity metrics. Building combined models increases the quality of the predictions, which is expressed by the correlations close to 0.60 in the SNA+METRICS experiment. The increase in the correlations values is significant at 99% (Wilcoxon signed-rank test).

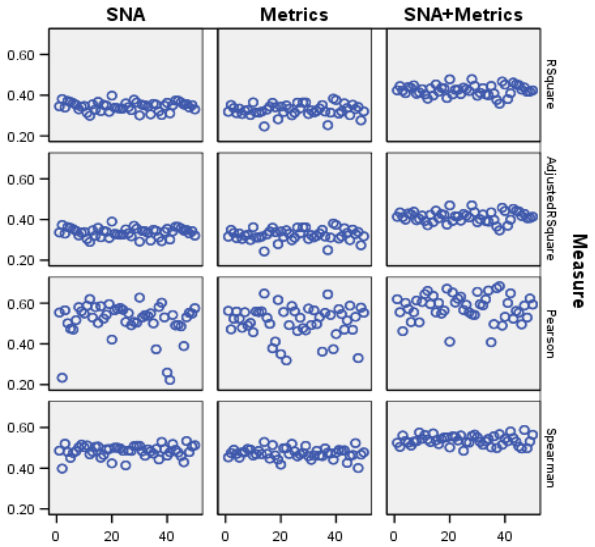


Figure 7. Results for linear regression

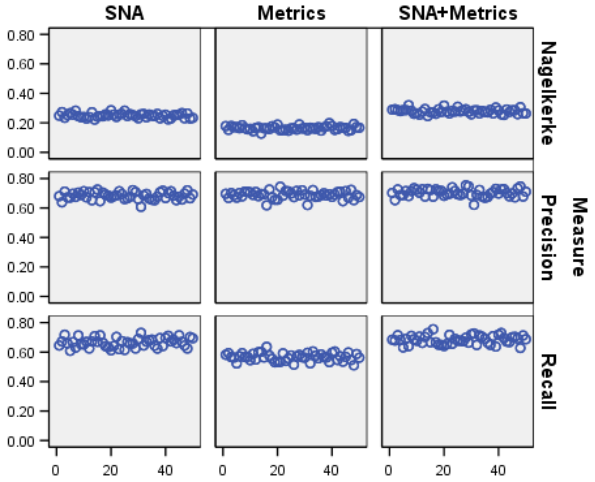


Figure 8. Results for logistic regression

BINARY LOGISTIC REGRESSION

We repeated our experiments using binary logistic regression model. In contrast to linear regression, logistic regression predicts likelihoods between 0 and 1. In our case, they can be interpreted as defect-proneness, i.e., the likelihood that a binary contains at least one defect. For training, we used the *sign(number of defects)* as dependent variable.

$$sign(number\ of\ defects) = \begin{cases} 1, & if\ number\ of\ defects > 0 \\ 0, & if\ number\ of\ defects = 0 \end{cases}$$

For prediction, we used a threshold of 0.50, i.e., all binaries with a defect-proneness of less than 0.50 were predicted as defect-free, while binaries with a defect-proneness of at least 0.50 were predicted as defect-prone.

In order to test the logistic regression models, we computed precision and recall. To explain these two measures, we use the following contingency table.

		Observed	
		Defect-prone	Defect-free
Predicted	Defect-prone (≥0.5)	A	B
	Defect-free (<0.5)	C	D

The *recall* A/(A+C) measures the percentage of binaries *observed* as defect-prone that were classified correctly. The fewer false negatives (missed binaries), the closer the recall is to 1.

The *precision* A/(A+B) measures the percentage of binaries percentage of binaries *predicted* as defect-prone that were classified correctly. The fewer false positives (incorrectly predicted as defect-prone), the closer the precision is to 1.

Both precision and recall should be as close to the value 1 as possible (=no false negatives and no false positives). However, such values are difficult to realize since precision and recall counteract each other.

Figure 8 shows the precision and recall values of the three experiments (SNA, METRICS, and SNA+METRICS) for logistic regression modeling. For each experiment, the values were consistent across the 50 random splits. The precision was around 0.70 in all three experiments. The recall was close to 0.60 for complexity metrics (METRICS), and close to 0.70 for the model built from network measures (SNA) and the combined model that used both complexity metrics and network measures (SNA+METRICS). These numbers show that network measures increase the recall of defect prediction by 0.10. Again, this increase is significant at 99% as measured by a Wilcoxon signed-rank test.

SUMMARY

The results for both linear and logistic regression support our hypothesis, that *network measures on dependency graphs, can predict the number of post-release defects (H3).*

5.4 THREATS TO VALIDITY

In this section we discuss the threats to validity of our work. We assumed that fixes occur in the same location as the corresponding defect. Although this is not always true, this assumption is frequently used in research [15, 30, 35, 41]. As stated by Basili et al., drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables [3]. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted.

We could not compare network measures against the *"best existing prediction technique",* simply because no technique has been emerge as the best so far. We chose complexity metrics as a benchmark, because they predicted defects in many past studies [2, 9, 35] and are readily available for most software projects.

Since this study was performed on the Windows operating system and the size of the code base and development organization is at a much larger scale than many commercial products, it is likely that the specific models built for Windows would not apply to other products, even those built by Microsoft.

This previous threat is often misunderstood as a criticism on empirical studies. Another misinterpretation is that nothing new is learnt from the result of empirical studies or more commonly "I already knew this result". Unfortunately, some readers miss the fact that this wisdom has rarely been shown to be true and is often quoted without scientific evidence. Further, defect data is rare and replication is a common empirical research practice.

We are confident that dependency data has predictive power for other projects—we will repeat our experiments for other Microsoft products and invite everyone to do the same for other software projects. Researchers interested in working with the Microsoft datasets are requested to get in touch with the authors to request access.

6. CONCLUSION AND CONSESQUENCES

We showed that network measures on dependency graphs predict defects for binaries of Windows Server 2003. This supports managers in the task of allocating resources such as time and cost for quality assurance. Ideally, the parts with most defects would be tested most.

The results of this empirical study are as follows.

- Complexity metrics fail to predict binaries that developers consider as critical (only 30% are predicted; Section 5.1).
- Network measures can predict 60% of these critical binaries (Section 5.1).
- Network measures on dependency graphs can indicate and predict the number of defects (Sections 5.2 and 5.3).
- When used for classification, network measures have a recall that is 0.10 higher than for complexity metrics with a comparable precision (Section 5.3).

We do not claim that dependency data is the sole predictor of defects—however, our results are another piece in the *puzzle of why software fails*. Other effective predictors include code complexity metrics [35] and process metrics like code churn [32]. In future work, we will build on the work by Briand et al. [10, 11] and identify more predictors and work on assembling the pieces of the puzzle. One of the missing pieces is the *human factor* [24] since humans are the ones who introduce defects. To reveal why programmers fail and build tools to prevent human failure is one of the big challenges for software engineering.

7. ACKNOWLEDGMENTS

We would like to thank Thirumalesh Bhat for his support on the MaX tool and Tom Ball, Marc Eaddy, Audris Mockus, Madan Musuvathi, and Andreas Zeller for discussions on the relationship between dependencies and defects. We would also like to thank the anonymous ICSE reviewers for their helpful comments.

8. REFERENCES

[1] F. B. e. Abreu and W. Melo, "Evaluating the Impact of OO Design on Software Quality," in *International Software Metrics Symposium*, Berlin, 1996.

[2] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object Orient Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* vol. 22, pp. 751-761, 1996.

[3] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," *IEEE Transactions on Software Engineering,* vol. 25, pp. 456 - 473, 1999.

[4] J. Bevan, "Software Instability Analysis: Co-Change Analysis Across Configuration-Based Dependence Relationships." PhD Thesis: University of California, Santa Cruz, 2006.

[5] J. Bevan and J. E. James Whitehead, "Identification of Software Instabilities," in *Working Conference on Reverse Engineering*, 2003, pp. 134-145.

[6] A. B. Binkley and S. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," in *International Conference on Software Engineering*, 1998, pp. 452 - 455

[7] D. Binkley and M. Harman, "An empirical study of predicate dependence levels and trends," in *International Conference on Software Engineering* 2003, pp. 330-339.

[8] S. P. Borgatti, M. G. Everett, and L. C. Freeman, "Ucinet for Windows: Software for Social Network Analysis," Analytic Technologies, Harvard, MA, 2002.

[9] L. Briand, P. Devanbu, and W. Melo, "An investigation into coupling measures for C++," in *International Conference on Software Engineering* Boston, Massachusetts, United States: ACM Press, 1997.

[10] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering,* vol. 25, pp. 91-121, 1999.

[11] L. C. Briand, W. L. Melo, and J. Wüst, "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects," *IEEE Transactions on Software Engineering,* vol. 28, pp. 706-720, 2002.

[12] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph " *Communications of the ACM,* vol. 16, pp. 575-577, 1973.

[13] R. Burt, *Structural Holes: The Social Structure of Competition*. Cambridge, MA: Harvard University Press, 1995.

[14] J. Cho, H. Garcia-Molina, and L. Page, "Efficient crawling through URL ordering," in *International Conference on World Wide Web* Brisbane, Australia: Elsevier Science Publishers B. V., 1998.

[15] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering,* vol. 26, pp. 797 - 814 2000.

[16] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 319 - 349 1987.

[18] R. A. Ghosh, "Clustering and dependencies in free/open source software development: Methodology and tools," *First Monday,* vol. 8, 2003.

[19] R. A. Hanneman and M. Riddle, *Introduction to social network methods*. Riverside, CA: University of California, Riverside 2005.

[20] A. E. Hassan and R. C. Holt, "The Small World of Software Reverse Engineering," in *Working Conference on Reverse Engineering*: IEEE Computer Society, 2004.

[21] S. M. Henry and D. Kafura, "Software Structure Metrics based on Information Flow," *IEEE Transactions on Software Engineering,* vol. 7, pp. 510-518, 1981.

[22] S.-K. Huang and K.-m. Liu, "Mining version histories to verify the learning process of Legitimate Peripheral Participants," in *International Workshop on Mining Software Repositories*, 2005.

[23] E. J. Jackson, *A Users Guide to Principal Components*. Hoboken, NJ: John Wiley & Sons Inc., 2003.

[24] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *Journal of Visual Languages & Computing,* vol. 16, pp. 41-84, 2005.

[25] B. Korel, "The program dependence graph in static program testing," *Information Processing Letters,* vol. 24, pp. 103-108, 1987.

[26] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? An empirical study of bug characteristics in modern open source software," in *Workshop on Architectural and System Support for Improving Software Dependability* San Jose, California: ACM Press, 2006.

[27] L. Lopez-Fernandez, G. Robles, and J. M. Gonzalez-Barahona, "Applying Social Network Analysis to the Information in CVS Repositories," in *International Workshop on Mining Software Repositories*, Edinburgh, Scotland, UK, 2004, pp. 101-105.

[28] G. Madey, V. Freeh, and R. Tynan, "The open source software development phenomenon: An analysis based on social network theory," *Americas Conference on Information Systems,* 2002.

[29] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science,* vol. 298, pp. 824-827, October 25, 2002 2002.

[30] K.-H. Möller and D. J. Paulish, "An empirical investigation of software fault distribution," in *International Software Metrics Symposium*, 1993, pp. 82-90.
[31] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering,* vol. 18, pp. 423-433, 1992.
[32] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *International Conference on Software Engineering*, St. Louis, MO, 2005, pp. 284-292.
[33] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 364-373.
[34] N. Nagappan, T. Ball, and B. Murphy, "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures.," in *International Symposium on Software Reliability Engineering*, 2006, pp. 62-74.
[35] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *International Conference on Software Engineering*, 2006, pp. 452-461.
[36] N. J. D. Nagelkerke, "A note on a general definition of the coefficient of determination," *Biometrika,* vol. 78, pp. 691-692, 1991.
[37] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review,* vol. 45, pp. 167-456, 2003.
[38] M. Ohira, N. Ohsugi, T. Ohoka, and K.-i. Matsumoto, "Accelerating cross-project knowledge collaboration using collaborative filtering and social networks," in *International Workshop on Mining Software Repositories* St. Louis, Missouri: ACM Press, 2005.
[39] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering,* vol. 22, pp. 886 - 894 1996.
[40] A. Orso, S. Sinha, and M. J. Harrold, "Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging," *ACM Transactions on Software Engineering and Methodology,* vol. 13, pp. 199 - 239 2004.
[41] T. Ostrand, E. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering,* vol. 31, pp. 340 - 355 2005.
[42] A. Pogdurski and L. A. Clarke, "A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions on Software Engineering,* vol. 16, pp. 965-979, 1990.
[43] G. Sabidussi, "The centrality index of a graph," *Psychometrika,* vol. 31, pp. 581-603, 1966.
[44] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," in *International Symposium on Empirical Software Engineering* Rio de Janeiro, Brazil, 2006.
[45] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Transactions on Software Engineering and Methodology,* vol. 10, pp. 209 - 254 2001.
[46] A. Srivastava, T. J., and C. Schertz, "Efficient Integration Testing using Dependency Analysis," Microsoft Research-Technical Report, MSR-TR-2005-94, 2005.
[47] R. Subramanyam and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects.," *IEEE Transactions on Software Engineering,* vol. 29, pp. 297-310, 2003.
[48] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology 2002.
[49] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press, 1984.
[50] T. Zimmermann and N. Nagappan, "Predicting Subsystem Defects using Dependency Graph Complexities," in *International Symposium on Software Reliability Engineering* Trollhättan, Sweden, 2007.

APPENDIX

Table 5. Pearson correlation values between the number of defects and centrality measures as well as complexity metrics. Correlations significant at 99% are marked by (**) and correlations significant at 95% are marked by (*). Correlations above 0.40 are printed in boldface.

	Pearson Correlation		
Ego Network	**In**	**Out**	**InOut**
Size	.208(**)	**.419(**)**	.234(**)
Ties	.190(**)	**.421(**)**	.242(**)
Pairs	.152(**)	**.424(**)**	.154(**)
Density	.110(**)	-.266(**)	-.336(**)
WeakComp	.187(**)	.051(*)	.178(**)
nWeakComp	.130(**)	-.201(**)	-.215(**)
TwoStepReach	.288(**)	.041	.051(*)
ReachEfficency	.155(**)	-.200(**)	-.226(**)
Brokerage	.152(**)	**.413(**)**	.153(**)
nBrokerge	.270(**)	.269(**)	.338(**)
EgoBetween	.156(**)	.265(**)	.164(**)
nEgoBetween	.198(**)	.329(**)	.290(**)
EffSize			.221(**)
Efficiency			.308(**)
Constraint			-.346(**)
Hierarchy			.208(**)
Global Network			
Eigenvector			.311(**)
Fragmentation			.261(**)
Betweenness			.265(**)
Information			.286(**)
Power			.367(**)
EffSize			.223(**)
Efficiency			.070(**)
Constraint			-.232(**)
Hierarchy			-.041
	Ingoing	**Outgoing**	**Symmetric**
Closeness	.005	.285(**)	.133(**)
Degree	.208(**)	**.419(**)**	.234(**)
dwReach	.302(**)	.252(**)	.133(**)
Complexity metrics		**Max**	**Total**
Functions			**.416(**)**
GlobalVariables			**.466(**)**
Lines		.243(**)	**.557(**)**
Parameters		.391(**)	**.533(**)**
FanIn		.345(**)	**.461(**)**
FanOut		.166(**)	**.480(**)**
Complexity		.049(*)	**.523(**)**
OO metrics		**Max**	**Total**
ClassMethods		.231(**)	.288(**)
SubClasses		.157(**)	.189(**)
InheritanceDepth		.218(**)	.185(**)
ClassCoupling		.224(**)	.210(**)
CyclicClassCoupling			.223(**)