Studying the impact of dependency network measures on software quality

Thanh H. D. Nguyen, Bram Adams, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: {thanhnguyen,bram,ahmed}@cs.queensu.ca

Abstract—**Dependency network measures capture various facets of the dependencies among software modules. For example, betweenness centrality measures how much information flows through a module compared to the rest of the network. Prior studies have shown that these measures are good predictors of post-release failures. However, these studies did not explore the causes for such good performance and did not provide guidance for practitioners to avoid future bugs. In this paper, we closely examine the causes for such performance by replicating prior studies using data from the Eclipse project. Our study shows that a small subset of dependency network measures have a large impact on post-release failure, while other network measures have a very limited impact. We also analyze the benefit of bug prediction in reducing testing cost. Finally, we explore the practical implications of the important network measures.**

I. Introduction

Activities such as unit test or code review require a great amount of time and effort. Software quality prediction allows software teams to prioritize these quality improvement activities. In particular, post-release failure prediction produces a priority list of high-risk modules that should be reviewed and tested to effectively catch as many potential bugs as possible. By testing the higher risk modules first, the team can catch more potential bugs in a shorter time frame.

Studies on commercial and open source projects have applied different types of metric to study software quality. Several studies [1]–[5] have shown that all too often process metrics, such as history of code changes or bugs reports, are better indicators of code quality over product metrics [6], [7] such as lines of code or fan-in.

In recent work, Zimmerman et al. [8] enhanced the performance of product metrics by incorporating metrics that are based on the dependency graph of a software system. Although some product metrics such as fan-in and McCabe's cyclomatic complexity already quantify some facets of the dependency structure of a software system, Zimmerman et al. explored over sixty metrics which were derived using concepts from the Social Network Analysis (SNA) field. By combining these SNA metrics with product metrics, Zimmerman et al. improved the prediction performance by 10% when studying the post-release failures of the Windows Server 2003 operating system.

Zimmerman et al.'s results are promising since their technique is one of the few that have helped improved the performance of product metrics. Such metrics require minimal knowledge of the history of a product. Thus they can be used for newly developed projects and for projects with no access to a well-archived history. However, a few important questions remain unanswered:

- RQ1: **Do the original findings generalize to other projects?** Recently, Tosun et al. [9] have shown that these findings do not hold for small projects. We wish to explore in detail whether these findings holds for Eclipse, a large open source project.
- RQ2: **Are the findings practically significant?** Zimmerman et al. have shown that these metrics lead to improvement in prediction performance but they did not explore the practical significance of their findings. A good understanding of the practical significance would encourage practitioners to adopt these new types of metrics.
- RQ3: **Why do SNA metrics improve the prediction performance?** Zimmerman et al. did not explore the rationale for the improved performance when using these metrics. A good understanding of the rationale would encourage practitioners to adopt SNA metrics. Practitioners are unlikely to adopt black-box like findings, instead they need to understand the rationale very well and ideally have it linked to their intuition.

In this paper, we seek to answer these three research questions by replicating the original study on a large open source project and performing a detailed analysis of the results to measure the practical significance of prediction models.

The main contributions of this paper are as follows:

- We confirm prior results on an industrial system using an open source system. SNA metrics do improve the prediction performance for quality models.
- We show that, although predicting bugs at the class level shows lower precision and recall than at the package level, the class-level predictions are more significant in practice than the package-level predictions.
- Our analysis shows that the performance of SNA metrics drops as they become less local. Metrics that capture the local neighbourhood of a file, i.e., ego network measures, contribute considerably to understanding the quality of a software system, more so than global metrics. These findings confirm the common software engineering wisdom concerning information hiding.

978-1-4244-8628-1/10/$26.00 ©2010 IEEE

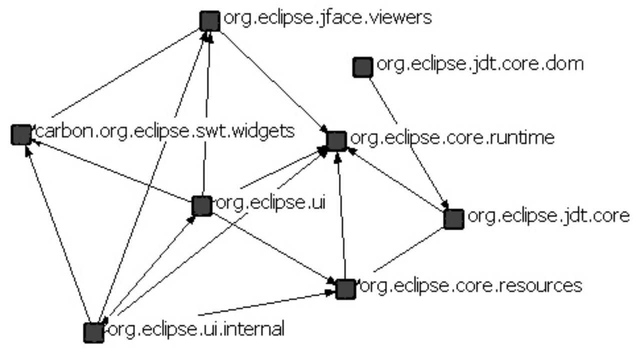


Fig. 1. Example of dependency network at the package level in Eclipse.

Paper organization. Section II defines the dependency network and the motivation behind using the SNA measures to predict post-release failures. We describe our study design in Section III. Then in RQ1, we describe and discuss the results of the replication. In RQ2, we study the practical implication of the prediction results. In RQ3, we describe our explanatory factor analysis to find which and how dependency network measures influence the prediction results. The related work is discussed in Section IV and threats to validity in Section V. We conclude in Section VI.

II. SNA MEASURES OF DEPENDENCY NETWORK

Figure 1 shows an example of a dependency network. The dependency networks of a software system are graphs whose nodes are the system's modules and whose edges are dependencies between the modules. Module A depends on module B if A calls B, A inherits from B, or A refers to a variable in B.

The use of dependencies for explaining programming errors has been discussed since the late 1970s [10]. Famous metrics such as Halstead's [11] or McCabe's cyclomatic [12] complexity are based on the dependencies between modules. However, there are many other network measures such as the ones used in social network analysis (SNA). Zimmermann and Nagappan [8] applied SNA measures on the dependency network and found that the prediction performance is better than using only the existing product metrics.

In the context of SNA, each module belongs to two types of network: an ego network and a global network. The ego network of a module consists of the module itself and every other modules that depend on it or whom it depends on. When an engineer modifies a class, he or she is most likely only concerned about the classes that the class calls or the classes that might need to call this class. The ego network of the class captures this view. On the other hand, the global network has a global scope. It is the network of all modules in the system.

Figure 1 shows the global dependency network of the packages in Eclipse. We only show the eight packages with the highest number of dependencies. The ego network for a package can be constructed by considering only the incoming, outgoing or a combination of both edges. In our example, the incoming ego network for *jface.viewers* consists of itself, *ui*, and *ui.internal*. The outgoing ego network consists of itself, *swt.widgets* and *core.runtime*. The undirectional ego network of *jface.viewers* consists of itself and all four other packages.

TABLE I
DESCRIPTION OF **EGO NETWORK** SNA MEASURES FOR EACH MODULE.

Measures	**Description**
	Descriptive
Size/Ties/Pairs	The number of modules, dependencies, unique pairs of modules occurring in the ego network
Density	# ties / # of possible ties
	Centrality
X2StepReach	# modules that are reachable in two steps / size
ReachEffic	X2StepReach normalized by size
Broker	# pairs of modules that are not directly connected to each other in the ego network
nBorker	Broker normalized by size
EgoBetween	# shortest paths between any two modules in the ego network / # paths pass through the ego module
nEgoBetween	EgoBetween normalized by size
	Structural holes
EffSize	# modules connected to the ego - average # ties between those modules
Efficiency	EffSize normalized by size
Constraint	Extent to which a module is limited in options to reach other modules in the ego network
Hierarchy	Concentration of constraint is in the ego network

TABLE II
DESCRIPTION OF **GLOBAL NETWORK** SNA MEASURES FOR EACH MODULE.

Measures	**Description**
	Centrality
Degree (Freeman cent.)	# modules that are directly connected to the module. This is the simplest centrality measure.
Eigen. cent.	Similar to degree centrality but also considers the neighbour's connectivity. Higher when a module is connected to many highly connected neighbours.
Power	Another extension of degree centrality. Similar to Eigen. cent. but is calculated differently.
Closeness	The total length of all shortest paths from the module to all other modules.
Betweenness	Similar to closeness centrality. Higher when the module occurs more often in shortest paths between any two modules in the network.
Information	Similar to closeness centrality. It is the harmonic mean of the length of all paths leading to the module. Smaller if the module is connected to many other modules through many shortest paths.
dwReach	The weighted number of modules that can be reached from the module. The weight is 1, 1/2, 1/3 for nodes that are 1, 2, or 3 steps away. Higher when there are many modules connected to the module via shorter steps.
	Structural holes
EffSize	# nodes connected to the module - average # ties between the nodes.
Efficiency	Eff. Size normalized by size of the network.
Constraint	Measures the extent to which the module is limited in options to reach other modules.
Hierachy	Concentration of constraint in the global network.

Table I describes the ego network measures, i.e., the measures expressing structural dependencies in the ego network of a module. For each module, we can compute each measure for the three types (in, out, and undirected) of ego network of the module. When we describe these measures in later sections, we use In, Out and Un to indicate the type of ego network the measure was computed on. For example, ReachEff In is the

TABLE III
DESCRIPTION OF COMPLEXITY METRICS [8].

Measures	Description
Module-related metrics	
GlobalVariables	# global variables
Function-related metrics	
Lines	# executable lines of code
Parameter	# arguments
FanIn	# other functions who calls this function
FanOut	# functions whom this function calls
Complexity	# McCabe's cyclomatic complexity [12]
Object oriented-related metrics	
ClassMethods	# methods in the class
SubClasses	# subclasses of the class
InheritanceDepth	Depth of the class in the inheritance tree
ClassCoupling	Coupling between this class and other classes

reach efficiency of the incoming ego network. Table II shows the global network measures. Similar to the ego network, some measures can be computed using only the incoming, outgoing, or both type of edges. We indicate this using In, Out, and Un.

Table III shows the traditional complexity metrics that were used in the original study. The metrics are computed on the functions of each module. For each module, we take both the total and the maximum value of its functions' measures. For example, if class A contains three functions B, C, and D with 13, 41, and 21 lines of code (LOC) respectively, then the total LOC of A is 75 and the max LOC of A is 41. We use both in the prediction. We put Total and Max next to the metric to indicate which calculation is used.

To avoid repetition, we will refer to the dependency network's SNA measures as SNA measures or measures in the rest paper. Similarly, we will use the complexity metrics or metrics for source complexity metrics. We also use ego measures for ego network's SNA measures and global measures for global network's SNA measures. More details on SNA measures can be found in [13].

III. STUDY DESIGN

To answer our research questions from Section I, we first replicate the original study [8] using the Eclipse project. Using the replication result, we examine the performance and the practical implication of the new prediction models. Then, we explore the prediction model using a hierarchical modelling technique to identify the high-impact SNA measures.

The level of analysis in our study is different from the original study. The modules in the original study are dynamically linked libraries (DLL). DLLs are shared libraries consisted of multiple object files on the Windows operating system. A DLL contains most of the executable code that is called from the main executable of the application (an .EXE file). When a failure is fixed inside a DLL a bug report is closed. The report identifies the DLL in which the bug occurred. This allows the original authors to link the bug back to the DLL. The Eclipse project does not have the notion of DLLs since it is implemented in Java. Java application has classes. Multiple classes are grouped into packages. We decide to replicate the study at both the class level and the package level. At the class level, the modules are classes. At the package level, the modules are packages.

The original study defined four sets of measures: dependency networks, source code metrics, post-release failures, and critical modules.

Post-release failures. The data for post-release failures is available for both the class and package level of Eclipse versions 2.0, 2.1, and 3.0 [14]. In this study, we will use only version 2.1 since the size of the data (5271 classes and 367 packages) is small enough for UCINET. UCINET is a 32-bit application so it cannot handle Eclipse 3.0 large data.

SNA measures. We download the binary release of Eclipse 2.1. We then extract the classes and packages from all of the plugins in the Eclipse release. Then, we use the Structure101 tool [15] to extract the dependencies such as function calls or imports connecting the classes and packages. We use the extracted data to construct dependency networks. Then, we use UCINET [16] to compute the SNA measures for each module in the network.

Complexity metrics. We download the source code release of Eclipse 2.1. We then determine the complexity metrics, as defined in the original study, of each class and package using a combination of the Understand tool [17] and some of the pre-calculated complexity metrics in the PROMISE dataset [14]. This results have 18 complexity metrics. The only metric we cannot compute is the CyclicClassCoupling, which is a patented metric [18].

Critical modules. In the original study, critical modules (called escrow binaries) are important DLLs on Windows. The critical modules are defined by the system experts. Eclipse developers do not maintain such a list. Instead, we derive the list of critical classes and packages based on the history of the project. We mark a class or a package as critical if there are post-release failures associated with the class or package in all three versions 2.0, 2.1, and 3.0 of Eclipse. This results in 74 critical classes and 93 critical packages.

RQ1: DO THE ORIGINAL FINDINGS GENERALIZE TO OTHER PROJECTS?

In RQ1, we want to find out if the original findings [8] can be generalized to other projects. We replicate the original study using data from the Eclipse project. We will verify the following hypotheses from the original study:

H1: Dependency SNA measures indicate critical classes/packages that are missed by complexity metrics.
H2: Dependency SNA measures positively correlate with post-release failures.
H3: Dependency SNA measures can be used to predict post-release failures at class and package level.

For each hypothesis, we (1) describe our approach, (2) present the findings, and then (3) analyze the results.

We note that Tosun et al. [9] verified the second part of H3 using the Eclipse project data. However, their results are different from ours because they predicted both pre-release and post-release failures together. The focus of their study was to improve the prediction performance, whereas we focus on understanding the prediction. Hence, we only predict post-

TABLE IV
AVERAGE RECALL OF PREDICTING CRITICAL CLASSES AND PACKAGES USING EACH SNA MEASURE AND COMPLEXITY METRIC. WE ONLY SHOW THE FIVE MEASURES AND METRICS WITH THE HIGHEST RECALL.

Class		**Package**	
SNA measures			
Degree in (g)	25.00%	Degree un (g)	59.69%
EgoBetween in	25.00%	Degree out (g)	59.69%
Pairs in	23.97%	Power	51.57%
Size in	23.97%	nWeakComp un	48.39%
Brower in	23.81%	EgoBetween out	47.69%
Complexity metrics			
Lines Total	35.29%	Parameters Total	60.36%
Comp. Total	33.33%	FanOut Total	60.00%
ClassMethods Max	32.46%	Lines Total	59.68%
ClassMethods Max	31.25%	Complexity Total	59.32%
FanOut Total	29.71%	FanIn Total	58.94%

release failures as in the original study.

H1: Dependency SNA measures indicate critical classes or packages that are missing by complexity metrics.

Approach. To check if SNA measures can indicate critical classes or packages missed by complexity metrics (H1), we use the same analysis as the original study. We randomly split the data into 2/3 for training and 1/3 for testing. Then, for each of the SNA measures, we build, train, and test a logistic regression model that tries to predict probability that the modules will be critical. We repeat this 50 times. Then, we select the modules with the highest predicted probability as critical. The number of modules picked is based on the number of actual critical modules. For example, for packages, we pick the 93/3=31 packages with highest probability because there are 93 critical packages and we use only 1/3 of the packages for testing.

The recall of a model based on a particular measure indicates how good the measure is as a predictor for critical modules. We calculate the recall by dividing the number of correctly predicted critical modules by the number of actual critical modules. We perform the same analysis for complexity metrics. Then, we compute the average recall of the 50 models for each measure and metric.

Findings. Table IV shows the average recall values for predicting critical classes and packages using each of the SNA measures (top) and complexity metrics (bottom). To save space, we only show the five measures and metrics with the highest recalls.

Analysis. In the original study, Zimmermann and Nagappan [8] showed that the top SNA measures recall twice as high as the top complexity metrics. Some of the SNA measures were able to reach recall values of around 60%, while the highest recalls for complexity metrics were around 30%. In our replication on Eclipse classes and packages, we observe that the top recall of the SNA measures are similar if not worse than those of the complexity metrics. The top recalls for SNA measures are 59.69% and 25.00% at the package and class level respectively. The top recall values for complexity metrics are 60.36% and 35.29% at the class and package level. Hence we reject H1.

H2: Dependency SNA measures positively correlate with post-release failures.

Approach. To test H2, we calculate the Spearman's rank coefficient between each SNA measure of a module and post-release failure of that module. The coefficient shows the strength of the correlation between the measure and the post-release failure. We use Spearman because the alternative, Pearson, requires that the two variables be normally distributed. Similar to the original study, we confirm that the Eclipse post-release failures are also not normally distributed. Hence we use the Spearman correlation. Both Pearson and Spearman yield a coefficient between -1 and 1. If the coefficient is -1, there is a 100% negative correlation between the two variables. If it is 1, then there is a 100% positive correlation between the two. 0 means that there is no correlation at all.

Findings. Due to space limitations, we cannot show all correlations here. However, we calculate the median of the absolute values of all correlations between the SNA measures and the number of post-release failures. The median correlation for the classes and packages are 0.0944 and 0.1542. We note that the medians can only be used for illustration only. It is not proper to make statistical inferences on these medians.

Analysis. Compared to the median correlation of 0.3195 for the DLLs from the original study, the correlation at the package level is overall lower. At the class level, the correlation is even less. The reason behind this is that the lower the level of analysis, the rarer the post-release failure. Only 11.99% of classes contain failures (632/5271) while 25.34% of the packages contain failures. When we lift the level of granularity from the class to the package level, we are essentially eliminating the classes that do not contain failures by grouping them with the ones that do. This explains why the correlation is much lower at the class level than at the package level. It also implies that prediction at the lower level, while potentially more valuable, is inherently less accurate than at the higher level.

In summary, we find that there is positive correlation between most of the SNA measures and the post-release failures. So we cannot reject H2. However, the correlation is low compared to that of the original study.

H3: Dependency SNA measures can be used to predict post-release failures at class and package level.

This hypothesis can be broken down into two sub-hypotheses:

- H3.1: SNA measures of a module can be used to predict the number of post release failures of the module.
- H3.2: SNA measures can be used to classify a specific module as high-risk or error-prone which means that the module has at least one failure.

The original study [8] found that a combination of SNA measures and complexity metrics can be used to predict the number of post-release failures (H3.1) and to predict error-prone modules (H3.2) for each DLL better than using

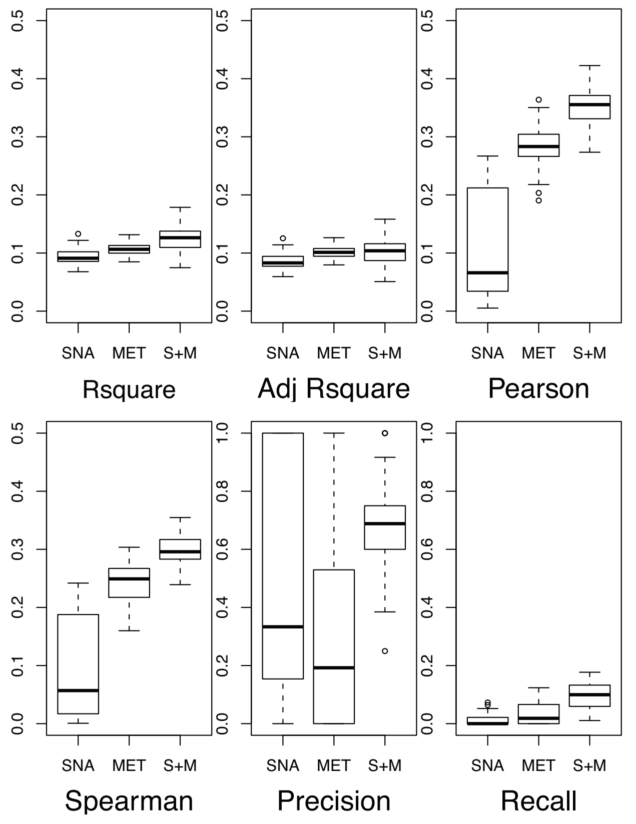


Fig. 2. H3.1 (Rsquare, Adj. Rsquare, Pearson, Spearman) and H3.2 (Precision, Recall) results for Eclipse at class level. The box plot shows the results of running the prediction on 50 random splits.

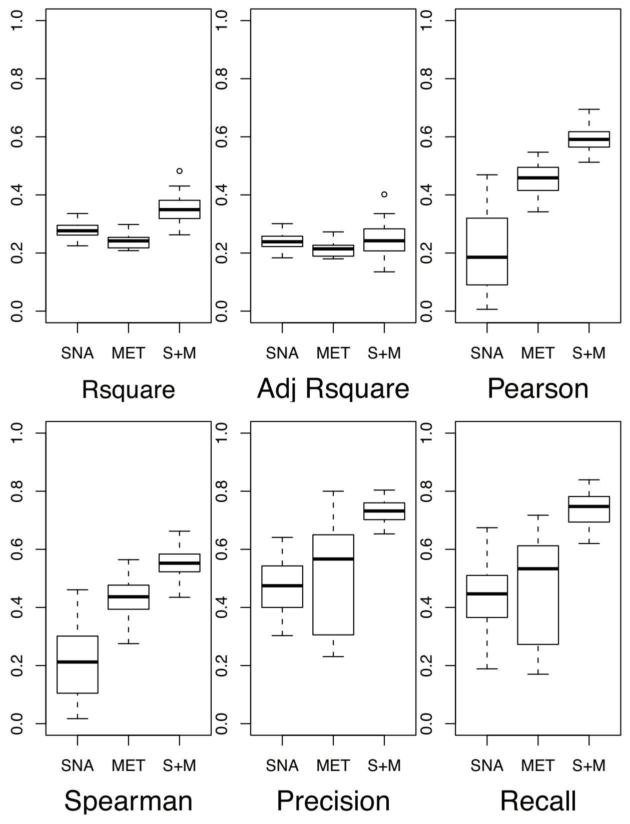


Fig. 3. H3.1 (Rsquare, Adj. Rsquare, Pearson, Spearman) and H3.2 (Precision, Recall) results for Eclipse at package level. The box plot shows the results of running the prediction on 50 random splits.

the complexity metrics alone. In H3.1, the original authors observed that the Spearman correlation between the predicted and actual number of post-release failures increases 10% when SNA measures are used together with complexity metrics instead of using the complexity metrics alone. In H3.2, There was a 10% increase in recall with comparable precision. The original authors concluded that SNA measure can be used to predict post-release failures.

Approach. To test H3.1 on the Eclipse data, we build linear regression models to predict the number of post-release failures. For H3.2 we build logistic regression models to predict the error-proneness of specific classes and packages. We build the models using the same method as presented in the original study. Similar to H1, we train the models using 2/3 randomly selected classes or packages and test them on the rest of the data. We ran the models on 50 different random splits. Because of the skewness of the data, as discovered in H1, we take the log of all measures and metrics. We also log take the log of post-release failures.

We report the test results as in the original study. For H3.1, we report the performance of the prediction model using four indicators: the Rsquare, Adjusted Rsquare, Pearson correlation, and Spearman correlation. Rsquare indicates how good the linear model fits the training data. Adjusted Rsquare has the same meaning as Rsquare but also takes into account the number of variables in the model. The higher the Rsquare and the Adjusted Rsquare, the better the fit. The Spearman and Pearson correlation coefficients show how good our prediction is. They show the correlation between the predicted and the actual number of post-release failures. The higher the correlation, the better the prediction. Since the data is not normally distributed, Spearman is more accurate than Pearson.

For H3.2, we build logistic regression models that predict the error-prone classes or packages. The models classify the module as error-prone if the probability that the module has at least one post-release failure is more than 50%. Similar to the original study, we report the Precision and Recall of the prediction on 50 random splits. The higher the Precision and Recall, the better the prediction. We note that the models in H3.2 are similar to the models in H1. However, the logistic models in H1 use each SNA measure individually. The logistic models in H3.2 use all SNA measures together. In addition, the models in H1 predict critical modules while the ones in H3.2 predict the existence of post-release failures.

As in the original study, we employ Principal Component Analysis (PCA) to remove collinearity in the data before using the data for prediction. When building linear models, researchers use independent variables, i.e., complexity metrics and SNA measures, to predict the dependent variable, i.e., post-release failures. However, if the independent variables are highly correlated, the prediction power of the model will be compromised. This is particularly true for prediction models in software engineering. For example, the number of LOC in a class will be highly correlated to the number of functions in that class. PCA is a common technique to eliminate this problem.

PCA combines all independent variables into principal components (PCs). We use the PCs to predict the post-release failures instead of the independent variables themselves. The PCs are ranked by the variance of the independent variables that they contain. The higher the variance captured by the PCs, the better the PCs represent the variables. If we use all the PCs, we will have 100% of the data variance. However, most studies would choose the first few PCs that are accountable for certain percentage of the variance to reduce the amount of variables used in the model. For example, when we build a model using only the SNA measures, the first five PCs are accountable for 66.6%, 13.0%, 9.93%, 3.61%, and 2.2% of the variance respectively. If we use the first five PCs, the model will contain 96.19% of all independent variables' variance. The original study used 95% as the threshold. We use the same threshold in our replication.

Findings. We present the results in Figure 2 for the classes and Figure 3 for the packages. Each box plot corresponds to each of the six indicators over the 50 runs. The first four (Rsquare, Adjusted Rsquare, Pearson, and Spearman) are for the linear model that predicts the number of post-release failures. The last two (Precision and Recall) are for the logistic model that predicts the error-proneness. For each box plot, the first column, SNA, is the result of only using the SNA measures. The second column, MET, is the result of only using the complexity metrics. The last column shows the result when we use both the SNA measures and the complexity metrics. We label this column S+M.

For H3.1, the Spearman correlation between the predicted and actual number of post-release failures improves by around 4.86% and 10.29% for the class and package level respectively when the SNA measures are used with the complexity metrics compared to using the complexity metrics alone. As for H3.2, the precisions improve by 30% and 25.2% at class and package level respectively. The recalls improve by 4.41% and 25.34%. All improvements are statistically significant using the Wilcoxon rank test at the 0.05 confidence level.

Analysis. H3.1. Our results show that the SNA measures improve the prediction of the number of post-release failures at both class and package level. In the original study, at the library level, the Spearman correlations increase for about 10%. In our replication, the improvement is lower classes (4.86%) but comparable for packages (10.29%). Hence, we cannot reject that SNA measures are suitable to predict the number post-release failures for packages. At the class level, the improvement is very low.

H3.2. For predicting the error-prone modules, we cannot reject the result of the original study at both the class and package level of Eclipse. The precision improves by 30% and 25.2% respectively. The recall improves by 4.41% and 25.34%. This is very similar to H3.1. However, the recall at the class level is very low.

TABLE V
SUMMARY OF THE RESULTS FROM THE ORIGINAL STUDY [8], AND FOR ECLIPSE PACKAGES AND CLASSES FROM OUR REPLICATION.

	DLL	**Pack.**	**File**
H1 SNA measures can predict critical modules that are missed by complexity metrics	Twice recall	Worse recall	Worse recall
H2 Average correlation between the SNA measures and the number of post-release failure*	28.45%	15.42%	9.44%
H3.1 Improves in # failures prediction when SNA measures are used with complexity metrics**	+10%	+10.29%	+4.86%
Actual Spearman correlation between predict and actual failures***	∼60%	55.23%	29.58%
H3.2 Improves in predicting error-prone binary when SNA measures are used with complexity metrics in precision and recall**	+∼0% +∼10%	+25.2% +25.34%	+30% +4.41%
Actual precision and recall***	∼70% ∼70%	73.19% 74.75%	68.83% 9.96%

(*) The average shown is the median of all correlations for illustration purpose only. It is not proper to statistically compare or make inference on these averages. (**) At 99% confident using Wilcoxon rank test. (***) We report the median values of all 50 random splits.

Summary of the replication

We summarize the replication results in Table V. We reject H1. We observe that the recall values when using the SNA measures are equivalent or worse than the recall values when using the complexity metrics. Thus, SNA measures miss more critical classes and packages than the complexity metrics. However, we note that Eclipse does not have a list of critical modules. We will discuss this threat to validity in Section V. For H2, we cannot reject that there is statistically significant correlation between the SNA measures and the number of post-release failures at both the class and package level. We also cannot reject H3, although the class level prediction results are very low.

RQ2: ARE THE FINDINGS PRACTICALLY SIGNIFICANT?

In the original study, the prediction recall and precision are very high for DLLs when SNA measures are used. Because the Eclipse project does not have artefacts similar to libraries, we replicate the study at lower levels: the package and the class level. We find that for H1 in RQ1, predicting critical modules is much less accurate at the class level than at the package level. We also find that the correlation between the metrics and measures and the number of post-release failures, in H2, is lower at the class level than at the package level. The correlations are both lower compared to the original study. In H3, it is clear that the prediction results are much worse at the class level than at the package level. A trend emerges from our replication: failure prediction at finer level of granularity is worse than at higher level granularity.

A. Practical significance of class vs package level prediction

Our first question is, given the low recall and precision at the class level, is the prediction at the package level more practical?

To understand how useful the prediction would be, we adapt the evaluation technique proposed by Mende and Koschke [19]

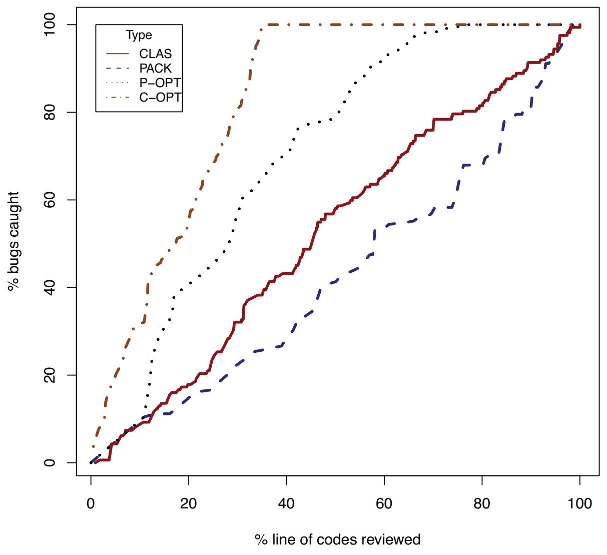


Fig. 4. Cumulative lift chart of the potential number of bugs discovered using the prediction at the class level (CLAS) and at the package level (PACK). We show the best case scenario for the class level (C-OPT) and the package level (P-OPT) for comparison. The prediction models here use both the new SNA measures and the traditional complexity metrics to predict post-release failure.

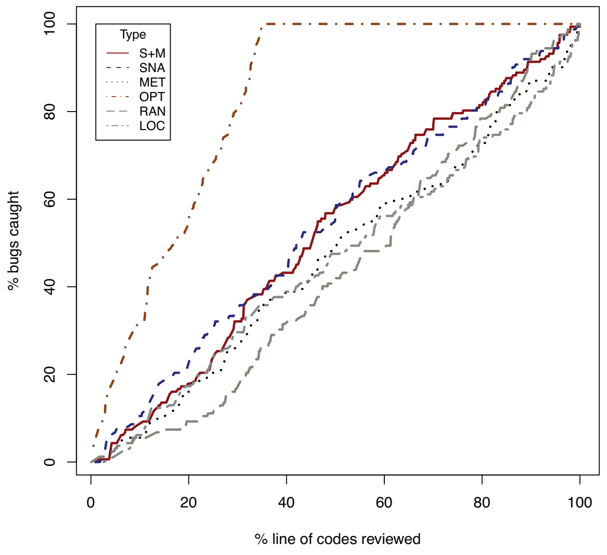


Fig. 5. Cumulative lift chart of the potential number of bugs discovered using different types of variable at the class level. The improvements by using S+M compared to using only the MET or the LOC are presented in Table VI.

to analyse the amount of effort needed over the practical overhead involved in of fixing bugs. We calculate the number of potential bugs found using the prediction at the class and the package level. We plot the result using a cumulative lift chart in Figure 4. The x-axis is the percentage of LOC that has to be reviewed. On the y-axis is the percentage of bugs discovered. The lines show the potential number of bugs that can be discovered by reviewing the corresponding LOC using

TABLE VI
PRACTICAL IMPROVEMENT WHEN SNA MEASURES ARE USED (S+M) COMPARED TO USING ONLY THE EXISTING COMPLEXITY METRICS (MET) OR THE LOC (LOC).

	% bugs caught				
% code	**S+M**	**MET**	**Diff**	**LOC**	**Diff**
10	8.64	5.56	3.09	6.17	2.47
20	17.90	16.05	1.85	17.28	0.62
30	32.10	25.93	6.17	29.63	2.47
40	43.21	38.89	4.32	38.89	4.32
50	58.02	49.38	8.64	47.53	10.49
60	65.43	58.02	7.40	56.17	9.26
70	76.54	62.96	13.58	61.79	14.81
80	81.48	72.84	8.64	74.07	7.41
90	91.36	86.42	4.94	82.10	9.26

prediction. We show the result for the class level (CLAS) and for the package level (PACK). Those two results are the best of the 50 random splits in H3.2. Figure 4 simulates the policy of testing higher risk modules first. We records the actual number of bugs, that we could have discovered, and the LOC of the module as we go along. Obviously, the smaller the percentage of LOC needed to discover a higher percentage of bugs, the better. The P-OPT line shows the optimal scenario at the package level. This is the best-case scenario when we are able to predict with 100% accuracy. In this case, we simulate testing the package with the highest number of actual failure first. The C-OPT is the optimal solution at the class level.

The first observation we can make is that the class level has much more potential to save practitioners effort than the package level. The C-OPT scenario only requires half the LOC to find all bugs compared to the P-OPT scenario (Figure 4). To discover all the bugs at the package level, the developers still have to inspect close to 80% of code in the best-case scenario compared to less than 40% at the class level. This is because a package contains a lot of code. If there is just one potential bug in a package, the developers have to inspect the entire package. At the class level, they would be able to focus on just the one class that contains the potential bug. This means that prediction at finer level of granularity brings more practical value than at higher levels i.e. package. In fact, this result shows that predicting failures at higher levels is not practically useful.

The second observation we can make is that, even with very low precision and recall, the class level prediction is more effective. This comes as a surprise because the recall and precision are 9.96% and 68.83% for classes compared to 74.75% and 73.19% for packages (see RQ1). These lower figures intuitively tell practitioners that the prediction is not useful at the class level. However, if we observe how higher the CLAS line is compared to the PACK line, we can see that for any given percentage of LOC that the team can inspect, the actual number of bugs discovered is almost always higher at the class level. This also suggests that future investigations should concentrate at improving the prediction at the class level, given the potential benefit in practice. Similar observation where noted by Mendes et al. [19] on basic prediction models.

B. Practical significance at the class level

Our second question is, concretely, how many more bugs are we catching with the aid of the new SNA metrics compared to only using the existing complexity metrics. We know from H3.2 that recall improves by 4.41% and precision improves by 30%. But what does it mean in practice?

Figure 5 shows the cumulative lift chart of the potential number of bugs caught using different types of variable in the prediction at the class level. This chart is similar to the one in Figure 4. The MET line shows the prediction that uses only the complexity metrics. The SNA line shows the prediction that uses only the SNA measures. The S+M line shows the prediction using both. These prediction models are the same as the ones used in H3 (see Figure 2 and 3). We also show the best case scenario OPT. This is the case where we can predict the post-release failure at 100% accuracy. The RAN line shows the random prediction. In this scenario, the team would randomly pick classes for reviewing or testing. The LOC line shows the prediction by inspecting the class with highest LOC first.

We can observe that the S+M line is higher than the MET line. This means that when the new SNA measures are used (S+M), the percentage of potential bugs caught increases compared to only using the existing complexity metrics (MET). We show the differences in Table VI. We can see that the improvement is about 1.85% to 13.58% more bugs caught. So applying the new prediction models will allow practitioners to discover about 1.85% to 13.58% more bugs compared to using traditional metrics.

RQ3: Why do SNA metrics improve the prediction performance?

At this point, we know that, when used together with traditional metrics, the SNA measures help improve the prediction. We also show improvements' practical significant. The only question left is why do SNA metrics improve the prediction performance?

Let us consider the complexity metrics and the SNA measures as four groups of variables. The first group, **cm.size**, contains the complexity metrics that are related to the size of the class such as GlobalVariables or Parameters. The second group, **cm.dep**, are the complexity metrics that are related to dependencies of a class such as FanIn or FanOut. The third and fourth group are the new ego network, **ego**, and global network, **global**, measures. As opposed to the size-related complexity metrics, the dependency-related metrics, cm.dep, consider the dependencies of the class. However, they only consider the relationship between the class itself and the surrounding classes. The ego measures, on the other hand, also consider the relationship among the neighbouring classes. The global measures take into account not only the neighbouring classes but also all the classes in the system. As we go from cm.size and cm.dep to ego and global, the influence on the variables goes from highly local to a global perspective. Our question is, where is the most useful knowledge about bug prediction lies: ego, i.e. local, or global networks. To answer

TABLE VII
VIF OF THE VARIABLES AFTER ITERATIVELY FILTERING THE TOP VARIABLE WITH HIGHEST VIF GREATER THAN 2.5.

Variable	VIF	Variable	VIF
cm.size		X2StepReach out	2.16
GlobalVariables Total	2.23	ReachEffic out	2.03
Parameters Max	1.24	nBroker out	1.43
ClassMethods Max	2.41	nEgoBetween out	1.47
cm.dep		Hierarchy out	1.55
Complexity Max	1.68	nWeakComp un	1.89
SubClasses Total	1.11	nEgoBetween un	1.66
InheritanceDepth Max	1.25	**global**	
FanIn Max	2.31	Eigenvector	2.43
ego		Fragmentation	1.11
Density in	1.83	Efficiency	2.18
nWeakComp in	2.12	Hierarchy	1.97
X2StepReach in	2.03	inCloseness	1.40
ReachEffic in	1.55	outCloseness	1.40
Hierarchy in	1.63		

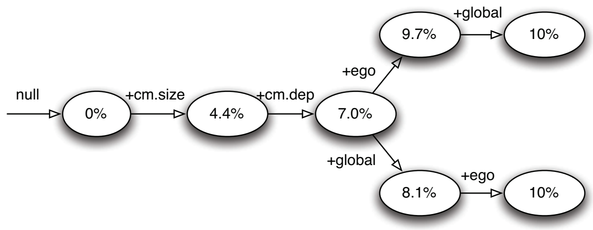


Fig. 6. Hierarchical modelling of the four groups of variables: **cm.size** - size-related complexity metrics, **cm.dep** - dependency-related complexity metrics, **ego** - ego measures, **global** - global measures. The numbers shown are the deviance explained ratio for each model.

this question, we use the same hierarchical modelling approach as in [4] to analyse the contribution of each group.

However, we first have to filter out variables with high degree of multicollinearity among the 19 complexity metrics and 64 SNA measures. In H3, filtering is unnecessary because the PCA alleviates multicollinearity. However, identifying the important variables is hard if they are distributed inside the PCs. So we have to run the RQ3 models without PCA. Thus we need to filter out the variables with high collinearity. A common indicator of multicollinearity is the Variable Inflation Factor (VIF). According to Allison [20], variables that have VIF greater than 2.5 are considered problematic. So we iteratively remove the variables with the highest VIF until the model does not contain any variable with VIF greater than 2.5. After this process, we are left with 3 cm.size, 4 cm.dep, 12 ego and 6 global variables. We report the VIF of the remaining 25 variables in Table VII.

Figure 6 shows the **deviance explained percentage** for each prediction model that we build hierarchically with the four groups of variables. This is the ratio between the deviance explained (DevEx) by the variable in the model and the deviance of a null model. The higher the DevEx, the better the model explains the independent variable (post-release failure). The DevEx for the model without any predictor is 0. When we add the cm.size variables, the DevEx increases to 4.5%. This means that the size-related complexity metrics explain about 4.5% of the post-release failures. When we add the cm.dep variables to the model, the DevEx jumps to 7%. That is a

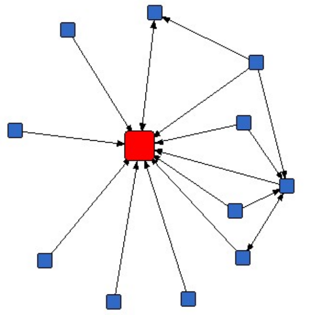
2.5% increase in explanation power. At this point, we first try to add the ego variables, then the global variables into the model (the upper path). We can observe that the DevEx increases by 2.7% when we add the ego measures and only 0.3% when we add the global measures. On the lower path, we first add the global variables, then the ego network. We can see that the DevEx only increases 1.1% for the global measures. But when we add the ego measures, the DevEx increases by 1.9%. The first conclusion here is that the SNA measures add as much explanation power, 3%, to the model as the dependency-related complexity metrics, 2.7%. The second result is that the ego measures explain most of the deviance brought in by the SNA measures. The global measures do not add much to the data as shown in the upper path. The lower path confirms this. We note that the low deviance explained is expected due to the rarity of the predicted event. The change of buggy class is about 12%. This low deviance still leads to practically significant predictive performance.

Our next question is which of the ego measures contribute the most to the prediction and what do that mean in practice. We perform analysis of variance (ANOVA) on each prediction model with all measures and metrics, i.e., the last one on the upper path of 6. ANOVA computes an F-test on each of the coefficients. The F-test shows whether a coefficient is statistically significantly different from zero. A zero or near zero coefficient in a linear model means that the variable does not effect the outcome of the prediction. Over all 50 random splits, ANOVA shows that only *nWeakComp in*, *X2StepReach in*, and *nEgoBetween out* are statistically significant at $p<0.05$. The coefficients for the three measures are all positive. They are the top contributors to the prediction model.

To demonstrate the meaning of *nWeakComp in*, *X2StepReach in*, and *nEgoBetween out*, we plot two examples of class dependency ego networks in Figure 7.

nWeakComp in is the number of unconnected components in the incoming ego network. The higher this number is, the higher the risk. What this says is that if many classes depend on the ego class and the classes are not connected to each other then the ego class has more risk of failure. We show an example in Figure 7(a). We believe this is an extension of FanIn. Popular interpretation of FanIn is that when many classes depend on a single class, there is a single point of failure. This increase the change of failure. nWeakComp adds to this interpretation that if all the dependent classes are completely independent for each other, then the risk of failure is higher than when they are interdependent. This is likely an indication that general library classes have higher chances of having bugs in them since they must serve a large number classes with varying expectations and goals.

X2StepReach in is the percentage of nodes in the global network that are indirectly connected to the ego class via its incoming connections within two steps. The higher the number, the higher the risk. We believe that this is another endorsement of the information hiding principle. If the software design allows proper encapsulation, the dependency graph should be deep and narrow [21]. In this case, the X2StepReach should be


(a) Incoming ego network of ant.core.AntCorePreference. Post-release failure = 3. *nWeakComp in* = 6.

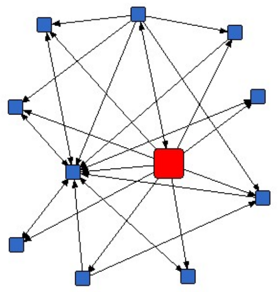

(b) Out ego network of ant.view.elements.ProjectNode. Post-release failure = 3. *nEgoBetween out* = 2.78.

Fig. 7. Examples of class level ego networks.

low. On the other hand, when every class is calling every other class, the X2StepReach is high. What this means in practice is that teams should pay attention when a class's X2StepReach is high. This may indicate that the design is deteriorating which may lead to post-release failures.

nEgoBetween out is the ratio of the number of all possible paths between the ego class's neighbours over the number of paths that pass through the ego. This measure is normalized by the size of the outgoing ego network. An example of a class with high *nEgoBetween out* is shown in Figure 7(b). We believe that this is another example of design deterioration. Very much like the example shown, the *nEgoBetween out* is usually high when one of the neighbouring classes is highly depended on by rest of the ego's neighbours which the ego is depended on. In Figure 7(b), the class on the left of the ego is an example of this situation since the ego class depends on many classes that also depend on this class. Our prediction models indicate that when this happens, the ego class is high-risk.

In summary, we find that among the new SNA measures, the ego measures are the most important factors. Within the ego measures, we find that *nWeakComp in*, *X2StepReach in*, and *nEgoBetween out* are the top three measures. We suggest the rationale behind each measure which are linked to common intuition in software practice.

IV. Related Work

The importance of verifying software engineering studies through replication has been emphasized in literature since the late 1980s by Basili et al. [22]. Daly et al. [23] replicated the famous Korson experiments [24] and found that modular code might not be a benefit to maintenance contrary to popular belief. This kind of outcome of a replication study motivates us to conduct a full replication of the original study [8] as presented in RQ1.

Research on software metrics has been a very active research topic since the original work by Halstead in the late 1970s [11]. In 1988, Cote et al. [25] surveyed the literature on software metrics and found more than 100 metrics. In recent years, with the new technologies in mining software repositories, researchers are able to study more about the relationship between software metrics and quality. Nagappan et al. [6], [7] used code complexity metrics, such as the ones in Table III, to predict post-release failure. More recent studies used process metrics such as the number of code commits [2] or even developer social structures [3]–[5] to predict quality.

V. Threats to validity

The main threat to validity of an external replication study like this one is construct validity. The goal of an external replication is to run the same analyses on a different dataset. If the data is not compatible it is hard to verify the validity of the replication data. For example, in the original study, critical DLLs used in H1 are determined by the developers through their experience. Because the Eclipse project does not define critical classes or packages, we have to build a list of critical modules based on the history of the project. This definition may not include all the critical classes or packages. Thus H1 might not be a valid replication. Also, since one of the metrics is patented, we cannot compute the metric for our replication. Unfortunately, we have no way of countering this threat.

We filter variables with high multicollinearity in RQ3. Otherwise, the prediction models will be very inaccurate and interpreting the models will be difficult. At the end of RQ3, we identify *nWeakComp in*, *X2StepReach in*, and *nEgoBetween out* as the three most important factors. However, variables that is highly correlated with the three factors have already been filtered out. To counter this threat, we check the correlations between the three variables and others. We found that for *X2StepReach in*, the only highly correlated variable (Spearman >0.7) is the same measures for undirectional ego network. For *nEgoBetween out*, it is the non normalized version. So they are not threats because the practical implication of these variables are the same. However, *nWeakComp in* is highly correlated with *nBroker in*. So we add *nBroker in* into the model an rerun ANOVA. The result shows that *nBroker in* is not significant. Thus it is not a problem either.

VI. Conclusion

In this study, we want to understand the roles of dependency network from SNA in prediction of post-release failures. We replicate the original study [8] using the Eclipse open source project. By studying the effort involved, we determine that bug prediction is more useful at the class level than at higher level. Using the models in the replication, we also find three SNA measures that have the highest impact on the prediction models. We linked the meaning of these measures to information hiding.

Our study suggests several possible research topics for future research. For instance, a different replication at the class level on a different project should verify if the three top SNA measures are still the most important ones and whether they still improve the failure prediction results. One should also investigate the benefits of predicting failures at a lower level than class, i.e., function level. However, this requires bug information at function level which is hard to collect.

References

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. of the 27th inter. conf. on Soft. Engi.* St. Louis, MO, USA: ACM, 2005, pp. 284–292.

[2] ——, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Proc. of the 1st inter. symp. on Empirical Soft. Engi. and Measurement*. IEEE Computer Society, 2007, pp. 364–373.

[3] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proc. of the 31st inter. conf. on Soft. Engi.* Vancouver, BC: IEEE Computer Society, 2009.

[4] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, 2009.

[5] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *Proc. of the 18th inter. conf. on Program Comprehension*. San Francisco, California, United States: IEEE Computer Society Press, 2010, p. to be appeared.

[6] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Early estimation of software quality using in-process testing metrics: a controlled case study," in *Proc. of the third workshop on Soft. quality*. St. Louis, Missouri: ACM, 2005.

[7] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. of the 28th inter. conf. on Soft. Engi.* Shanghai, China: ACM, 2006, pp. 452–461.

[8] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. of the 30th inter. conf. on Soft. Engi.* Leipzig, Germany: ACM, 2008, pp. 531–540.

[9] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proc. of the 5th inter. conf. on Predictor Models in Soft. Engi.* Vancouver, British Columbia, Canada: ACM, 2009.

[10] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, 1981.

[11] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[12] T. J. McCabe, "A complexity measure," in *Proc. of the 2nd inter. conf. on Soft. Engi.* San Francisco, California, United States: IEEE Computer Society Press, 1976.

[13] R. A. Hanneman and M. Riddle, "Introduction to social network methods," 2005, published in digital form at http://faculty.ucr.edu/ hanneman/.

[14] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. of the 3rd inter. workshop on Predictor Models in Soft. Engi.* IEEE Computer Society, 2007.

[15] Headway Software, "Structure101," 2009.

[16] Analytic Technologies, "Ucinet 6: Social network analysis software," 2007.

[17] Scientific Toolworks Inc., "Understand," 1996.

[18] N. Nagappan and T. Bhat, "Technologies for code failure proneness estimation," Patent, April 26, 2007.

[19] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. of the 5th inter. conf. on Predictor Models in Soft. Engi.* Vancouver, British Columbia, Canada: ACM, 2009, 1540448 1-10.

[20] P. D. Allison, *Multiple regression: A primer*. Thousand Oaks, CA: Pine Forge Press, 1999.

[21] K. J. Lieberherr and I. M. Holland, "Assuring good style for object-oriented programs," *Software, IEEE*, vol. 6, no. 5, pp. 38–48, 1989.

[22] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. Softw. Eng.*, vol. 12, no. 7, pp. 733–743, 1986.

[23] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "An external replication of a korson experiment," Empirical Foundations of Computer Science, Tech. Rep., 1994.

[24] T. D. Korson and V. K. Vaishnavi, "An empirical study of the effects of modularity on program modifiability," in *First workshop on Empirical studies of programmers*. Washington, D.C., United States: Ablex Publishing Corp., 1986, pp. 168–186.

[25] V. Cote, P. Bourque, S. Oligny, and N. Rivard, "Software metrics: an overview of recent results," *J. Syst. Softw.*, vol. 8, no. 2, pp. 121–131, 1988.