An Empirical Study on the Relation between Dependency Neighborhoods and Failures

Thomas Zimmermann[*], Nachiappan Nagappan[*], Kim Herzig[‡], Rahul Premraj[§] and Laurie Williams[†]
[*]*Microsoft Research, Redmond, WA 98052, USA · tzimmer,nachin@microsoft.com*
[‡]*Saarland University, Saarbrücken, Germany · herzig@cs.uni-saarland.de*
[§]*VU University Amsterdam, The Netherlands · rpremraj@cs.vu.nl*
[†]*North Carolina State University, Raleigh, NC 27695, USA · williams@csc.ncsu.edu*

***Abstract*—Changing source code in large software systems is complex and requires a good understanding of dependencies between software components. Modification to components with little regard to dependencies may have an adverse impact on the quality of the latter, i.e., increase their risk to fail. We conduct an empirical study to understand the relationship between the quality of components and the characteristics of their dependencies such as their frequency of change, their complexity, number of past failures and the like. Our study has been conducted on two large software systems: Microsoft VISTA and ECLIPSE. Our results show that components that have outgoing dependencies to components with higher object-oriented complexity tend to have fewer field failures for VISTA, but the opposite relation holds for ECLIPSE. Likewise, other notable observations have been made through our study that (a) confirm that certain characteristics of components increase the risk of their dependencies to fail and (b) some of the characteristics are project specific while some were also found to be common. We expect that such results can be leveraged for use to provide new directions for research in defect prediction, test prioritization and related research fields that utilize code dependencies in their empirical analysis. Additionally, these results provide insights to engineers on the potential reliability impacts of new component dependencies based upon the characteristics of the component.**

***Keywords*-software quality; defects; dependency; empirical software engineering**

I. Introduction

Dependencies between software components are widespread in software systems and are crucial for their successful operation. They are sometimes formed for strategic reasons such as to realize certain functionalities in the system and enable software reuse, or they can simply be side effects of the organizational structure [1]. Dependencies essentially paint a picture of information flow within a software system and exert at least some influence on the overall success and quality of the product. In the past, researchers have leveraged them for defect prediction [2], test prioritization [3] and investigating the intersection of social and technical congruence factors [4] and the like.

In this paper, we seek to extend our understanding of dependencies between software components by investigating the impact on the quality of a component (i.e., its likelihood to have a defect) by the state of its neighborhood (dependents and dependees, see below for definitions). For instance, consider two components *A* and *B* where *A* is dependent on *B*. If component *B* undergoes significant amount of code churn (added, deleted, or modified code) between revisions, it is reasonable to expect that component *A* will also undergo some change to be kept in sync with *B*. Thus, the effects of churn may propagate across dependencies and in turn, impact their quality by introducing new faults in them. Our conjecture is that the quality of a component may be influenced by the following characteristics of its dependencies: size, code churn, complexity, test coverage, and organizational structure.

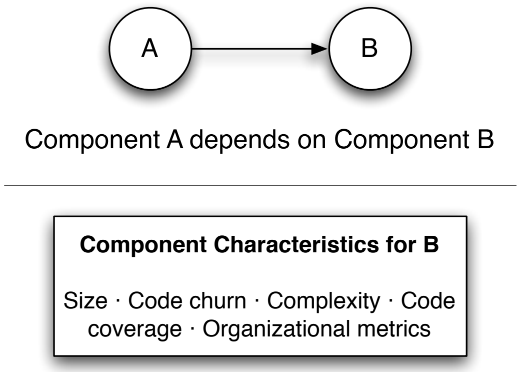

A
B
Component A depends on Component B
Component Characteristics for B
Size · Code churn · Complexity · Code coverage · Organizational metrics

Figure 1. Illustration of dependency relationships between Components *A* and *B*

Our conjecture was strengthened by a preliminary survey on developers at Microsoft whose views on the influence of neighboring components conformed to ours (Section II). Recognizing the importance of the role of dependencies in overall software quality, we thus conducted a systematic quantitative investigation on the relationship between the quality of a component and the characteristics of its neighboring components — this is the main contribution of our paper. But before presenting the details of our study, we define some of the terminology used henceforth in the paper:

- **Failure-proneness:** We consider failure-proneness as a measure of quality of a software component. Failure-

Table I
RESPONSES FROM A SURVEY ON MICROSOFT DEVELOPERS ON THE RISK OF FAILURE OF A BINARY BECAUSE OF THE CHARACTERISTICS OF ITS NEIGHBORING BINARIES.

		Risk of failure of binary				
Characteristics	Binary has a dependency with another that	Increases	Has no effect	Decreases	No opinion	Score
Size	has many methods	36	65	1	7	0.34
Churn	has churned a lot (quantity of LOC)	91	12	0	6	0.88
	has been changed many times	88	18	1	2	0.81
Code metrics	is complex	91	14	0	3	0.87
	calls many methods of other binaries	84	20	0	5	0.81
	is tightly coupled to other binaries	74	26	1	7	0.72
	has deep inheritance structures	57	31	3	18	0.59
	is called by many methods in other binaries	27	36	39	7	-0.12
Code coverage	has high test coverage	1	20	84	4	-0.79
People measures	has no clear owner	86	19	0	2	0.82
	was developed by ex-engineers	72	32	1	4	0.68
	has been developed by many engineers	60	34	3	12	0.59
	is not owned by your team	58	46	0	5	0.56
	is owned by your team	3	39	65	2	-0.58
Post-release failures	failed many times in the past	99	7	1	2	0.92

proneness is the probability that a component will fail in operation in the field. The higher the failure-proneness, the higher is the probability of experiencing a post-release failure. For our purposes, we treat all components known to have at least one post-release failure as failure-prone (FP).

- **Component Characteristics:** Component characteristics are descriptive attributes of a component that can differentiate it from other components. In this paper, the component characteristics of interest are size, churn, complexity, test coverage, and organizational attributes as shown in Figure 1 for component *B*.
- **Incoming dependency:** A component has an incoming dependency if syntactically another component utilizes its data or functionality. In Figure I, component *B* has an incoming dependency from component *A*.
- **Outgoing dependency:** A component has an outgoing dependency if syntactically it utilizes data or functionality of another component. In Figure I, component *A* has an outgoing dependency on component *B*.
- **Dependant:** A component is a dependent with respect to another component if it has an outgoing dependency on that component. In Figure I, component *A* is a dependant of component *B*.
- **Dependee:** A component is a dependee with respect to another component if it has an incoming dependency from that component. In Figure I, component *B* is a dependee of component *A*.

Our investigation involved the study of the relationship between the failure-proneness of components and the following characteristics of dependent and dependee components: size, frequency and degree of churn, complexity metrics, test coverage, and organizational attributes (e.g., number of developers involved). Additionally, we also investigated whether there is any relation between post-release failures of a component and the post-release failures of its dependants and dependees.

We conducted our study on two large software projects: Windows VISTA and ECLIPSE (versions 2.0, 2.1, and 2.2). The choice of projects allowed us to reflect on the generality of the relationship across different types of projects. While VISTA is written in the .NET framework, ECLIPSE is developed in the JAVA programming language. Also, the former is a commercial software and follows markedly different development processes in comparison to ECLIPSE, which is free and open-source. In the following sections, we present the details regarding collecting the necessary data from these two projects for our investigation (Section III), the experimental methodology (Section IV), followed by the results and their discussion (Section V). Next, we present some related work (Section VI) and conclude our paper with discussing the threats to validity (Section VII) and consequences and ideas for future work (Section VIII).

II. SURVEY AT MICROSOFT

In order to gauge whether developers share our opinion about the influence of neighboring components on a component's quality, we undertook a survey and invited 700 developers at Microsoft to participate in it. The questions in the survey were formulated in alignment to the characteristics mentioned above (or see column one in Table I). The complete list of questions are presented in the second column of Table I.

It is important at this point to note that when talking of components in Microsoft products, we are essentially referring to binaries. A binary comprises of several files complied together and is the lowest level to which post-release failures can be accurately mapped. When a post-release fix is performed, it usually involves changing several files that are complied into one binary. In the case of ECLIPSE, post-release failures can be mapped to files and hence we conduct our investigation at the file-level for the project, but refer to them as components in the paper.

We received 110 (15.7%) responses to our survey. The experience of the responders in terms of time spent in the software industry was ten years, while the median time spent at Microsoft was seven years, which is overall a very experienced population. The responses are compiled and presented in Table I. The third column in the table indicates the number of responders who consider that the characteristic (in the second column) increases the risk of failure of a depending binary. Likewise, the fourth and fifth columns indicate the number of developers who consider the characteristics to pose no risk or even decrease risk respectively. Column six presents the number of developers who had no opinion regarding the corresponding characteristic, while the last column is the overall score for each characteristic computed as ratio of the absolute difference between columns three and five to the sum of columns 3–5.

The responses suggest that few developers regard component size to have any bearing with the failure-proneness of depending components, while a substantial majority regarded degree of code churn to have a strong influence. Complexity metrics such as complexity, number of calls to other binaries, and coupling were regarded by many to increase the risk of failure, but incoming calls to binaries were to reduce the risk. Most developers also believe that better tested components put depending components at lower risk. A strong agreement was also noted on the influence of organizational measures on failure-proneness — lack of clear or self-team ownership and development by many or even ex-engineers increase failure-proneness. Lastly, nearly all developers considered that components with a number of past failures increase risk in the future.

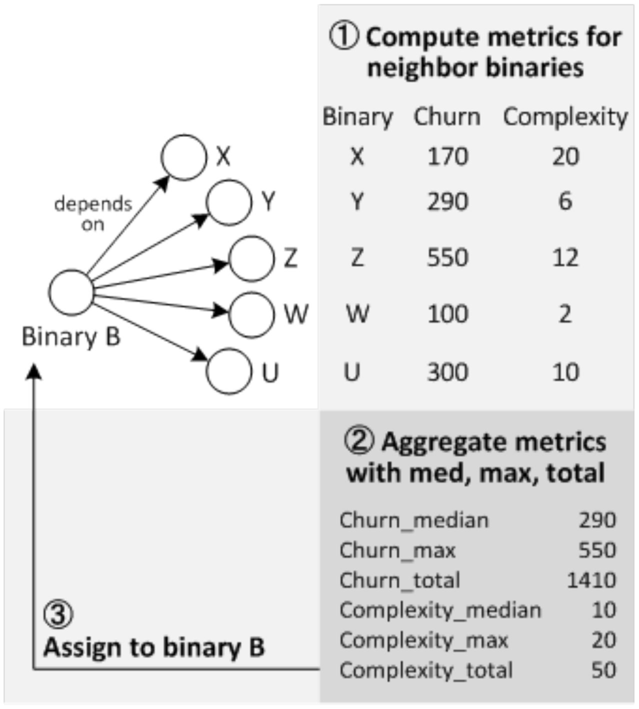
We were encouraged by the survey responses to carry out further investigation in this direction of work. The remainder of the paper is dedicated towards quantitatively investigating the relationships by examining two software projects.

III. Data Collection

In this section, we discuss our data collection process and delineate the metrics collected for the study.

A. Process

Figure 2 illustrates the process used to collect data for our study. In both projects, we first identify the dependents and dependees for each component; we call this the neighborhood of the component. The neighborhood strictly contains only the direct dependants and dependees. For example in Figure 2, component B has outgoing dependencies on components X, Y, Z, W, and U, i.e., the latter are dependees of component B and together comprise the *outgoing neighborhood* of B. Vice versa, component B also may have incoming dependencies which comprise its *incoming neighborhood*. For simplicity sake, Figure 2 only illustrates outgoing dependencies on component B. Together, incoming and outgoing dependencies of component B comprise its entire neighborhood.



① Compute metrics for neighbor binaries
depends on
Binary B
X
Y
Z
W
U
② Aggregate metrics with med, max, total
③ Assign to binary B

Binary	Churn	Complexity
X	170	20
Y	290	6
Z	550	12
W	100	2
U	300	10

Churn_median	290
Churn_max	550
Churn_total	1410
Complexity_median	10
Complexity_max	20
Complexity_total	50

Figure 2. For Binary B we identify the binaries it has a dependency with (neighborhood), compute metrics on the neighbors, compute the median values and assign them to Binary B.

Once the neighborhoods for each component have been identified, we compute the relevant metrics for each neighbor (see Section III-C for the list of metrics). To exemplify, in order to compute the values for two metrics – churn and complexity – for component B, we first compute the metrics each outgoing neighboring component and then compute the median of the values. We favored the median over the mean because it is more robust in the presence of outlier values. We then assign the median to component B as a characterization of B's outgoing neighborhood. The same data collection process is repeated for all metrics and incoming dependencies as well.

B. Tools and Methods

Microsoft Vista. The differences in the two projects required us to collect data using different tools and methods. To identify dependencies in VISTA, we used the Max tool [5] that generates a system-wide dependency graph from both native x86 and .NET managed binaries. Max tracks dependency information at the function level and includes dependencies that result from calls, imports and exports, remote procedure calls (RPC), COM, and accesses to the Windows Registry.

Eclipse. In the case of ECLIPSE, we first tracked class dependencies using Java byte code analysis to detect incoming and outgoing method calls and object usages. Collecting dependency data on byte code profits from type resolution performed by the Java compiler and thus is more accurate than analyzing source code. But this gain in accuracy comes at a cost – compiled class files do not necessarily have the same name as the source file in which the class was declared. Declaring two top-level classes A and B in one source file `A.java` and compiling it will result in two class files `A.class` and `B.class`. Hence, the dependencies are computed at the class level and not at the file level; we require the latter since post-release failures are mapped to files and not classes. Hence, in order to map classes to the files in which they have been declared, we parsed source code to detect which classes were declared in which source file. With this mapping in hand, we could reconstruct dependencies between files in the ECLIPSE project.

The metrics for ECLIPSE were obtained project from the publicly available data repository [6].

C. Computed Metrics

The metrics computed from the neighborhood components covered a variety of characteristics such as size, churn metrics, code metrics, coverage metrics, organizational metrics, and failure metrics. In our study, we separately analyzed each metric for both incoming and outgoing neighborhoods.

Below, we list the metrics collected from the two projects. Due to the differences between the projects and the availability of data, some metrics were specific to either project. Also, recall that the metrics were computed for VISTA at the binary level, while they were computed for ECLIPSE at the file level. This explains for many of the differences in the set of metrics computed from the projects. In the list below, we indicate whether the metric was computed only for VISTA, ECLIPSE, or for both.

Size. For each component, we determined the number of executable, non-commented lines of code (LOC) for VISTA and several more metrics for ECLIPSE (see Table III).

Churn. The churn metrics were collected relative to the previous released version, that is, for VISTA the baseline used for churn measurement was Windows Server 2003 SP1. Similarly for ECLIPSE, the churn metrics were computed using the previous releases as a benchmark. We computed the following churn metrics:

- **Frequency total:** We counted the total number of edits/check-ins that account for all the added, modified, and deleted LOC.
- **Churn size total:** We summed up the added, modified, and deleted LOC.
- **Relative churn size:** We derive relative churn of the extracted metrics (added, deleted, and modified LOC) as normalized values obtained during the development process. This measure quantifies the extent of overall work done in a file/binary per check-in. Prior work by Nagappan and Ball [7] showed that relative code churn measures are significantly stronger predictors of defect density in the Windows Server 2003 system than absolute code measures. They found that 89% of defect-prone binaries in Windows Server 2003 can be identified using relative code churn measures.

Code metrics. Several procedural and object-oriented complexity metrics were collected from the projects. In our analysis, we include both the total and maximum value for each of these metrics for VISTA. In the case of ECLIPSE, also computed the average of the metrics.

- **FanIn:** FanIn is the number of other functions calling a given function in a module (VISTA only).
- **FanOut:** FanOut is the number of other functions being called from a given function in a module (both VISTA and ECLIPSE).
- **Cyclomatic complexity:** The cyclomatic complexity metric [8] measures the number of linearly independent paths through a program unit (both VISTA and ECLIPSE).
- **Methods:** Number of methods in a class including public, private and protected methods (both VISTA and ECLIPSE).
- **Inheritance Depth:** Inheritance depth is the maximum depth of inheritance for a given class (VISTA only).
- **Coupling:** This metric signifies coupling to other classes through (a) class member variables; (b) function parameters; (c) classes defined locally in class member function bodies; (d) immediate base classes; and (e) return type (VISTA only).
- **No. of classes:** This is the count of the number of classes in the component (ECLIPSE only).
- **No. of interfaces:** This is the count of the number of interfaces in the component (ECLIPSE only).
- **No. of parameters:** This is the count of the total number of parameters in the component (ECLIPSE only).
- **No. of fields:** This is the count of the total number of fields in the component (ECLIPSE only).
- **No. of static methods and fields:** This is the count of the total number of static methods and static fields in

the component (ECLIPSE only). The two metrics were counted separately.

- **Nested block depth:** This is the measure of the depth of code blocks found in the component (ECLIPSE only).

Code Coverage. The following two metrics reflect the extent of test coverage in VISTA. Note that this data could not be reliably computed for ECLIPSE given the data publicly available and hence was only computed and used for VISTA.

- **Block coverage:** A (basic) block is a set of contiguous instructions (code) in the physical layout of a binary that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions. The number of blocks covered during testing constitutes the block coverage measure.
- **Arc coverage:** Arcs between blocks represent the transfer of control between basic blocks due to conditional and unconditional jumps, as well as due to control falling through from one block to another. Similar to block coverage the proportion of arcs covered in a binary constitute the arc coverage. Arc coverage can also be called branch coverage.

People measures. The following metrics relate to the organizational structure and developers of the product. Again, note that this data could not be reliably computed for ECLIPSE given the data publicly available and hence was only computed and used for VISTA.

- **Organizational Level:** The level in the organization structure of an organization at which the ownership of a binary is determined/attributed to a particular engineer [9].
- **Engineers:** The number of engineers to wrote/contributed code to a binary.
- **Ex-engineers:** The number of engineers who wrote/ contributed code to a binary who are no longer working for Microsoft.

Post-release failures. Post-release failures is the count of the number of fixes that were mapped back to components after the products were released for a time period of the first six months. We mapped post-release failures to the respective components associated with the failure.

In total, we computed 22 metrics from components for VISTA and each metric was computed for outgoing and incoming dependencies separately giving us a total of 44 metrics. In the case of ECLIPSE (2.0, 2.1, and 3.0) , we computed 252 distinct metrics and a total of 504 metrics accounting for both outgoing and incoming dependencies.

IV. Experimental Setup

Our experiments involves checking whether each metric (size, churn, code metrics, coverage, people attributes, and post-release failures) for neighboring components is significantly higher (or lower) for FP components compared to not failure-prone binaries (NFP, see Figure 3 for illustration in the VISTA context). For this we create two groups: the FP group that contains the values of the metrics for failure-prone components; and the NFP group that contains the values for components without any known failures. We then compare the distributions of the metrics in the two groups separately for incoming and outgoing neighborhoods using a one-tailed Mann Whitney-U test with the significance level was set to .05. Our motivation to examine the neighborhoods separately was to allow observation of any effects of each class of neighborhoods on the metrics. We ran two tests for each component with the following two hypotheses:

- **Hypothesis 1** Metric values for the FP group are significantly greater than for the NFP group.
- **Hypothesis 2** Metric values for the FP group are significantly lower than for the NFP group.

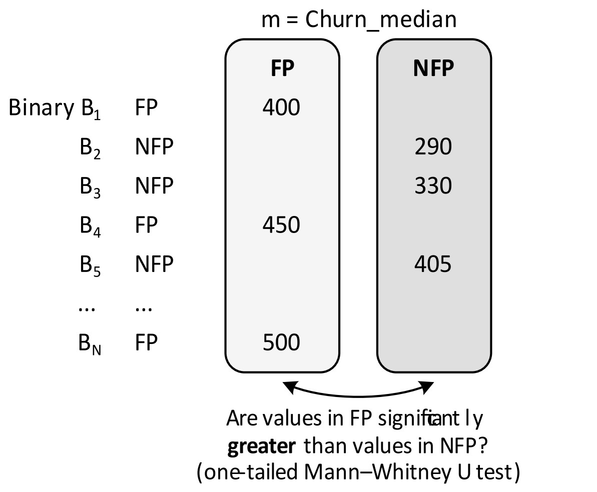


Figure 3. We statistically test for each metric (for example Churn_median) to determine if its values are significantly higher or lower for FP binaries compared to NFP binaries.

Given our experimental design, we accounted for multiple hypotheses testing using the Bonferroni correction. Compared to other correction techniques such as Benjamini-Hochberg, the Bonferroni correction is more conservative (less likely to reject null hypotheses) and thus less likely to accidentally accept incorrect results [10]. In total, we ran 88 tests for VISTA (two tests for each of the 44 metrics) and 504 for ECLIPSE (two tests for each of the 252 metrics). Upon applying Bonferroni correction, the significance values for the tests were revised to .00055 (.05/88) for VISTA and .00001 (0.05/504) for ECLIPSE.

V. Results

The results of our experiments are presented in Tables II and III. We compare the distributions of the median values for each metric for both incoming and outgoing dependencies separately. In the tables, the symbol $\triangledown$ indicates

Table II
RESULTS FROM MICROSOFT VISTA INDICATING SIGNIFICANT DIFFERENCES IN THE DISTRIBUTION OF METRICS ACROSS FP AND NFP COMPONENTS.

	Neighboring medians	
Metric	Incoming	Outgoing
Size		▽
Churn		
Churn Frequency	▲	▽
Churn Size		▽
Relative Churn		
Code metrics		
Fan in (max)		▽
Fan in (total)		▽
Fan out (max)		▽
Fan out (total)		▽
Cyclomatic complexity (max)		▽
Cyclomatic complexity (total)		▽
Class methods (max)		
Class methods (total)		▲
Inheritance depth (max)		▲
Inheritance depth (total)		▲
Class coupling (max)		▲
Class coupling (total)		▲
Code coverage		
Block coverage (total)		
Arc coverage (total)		
People measures		
Org. level	▲	
Editing engineers	▲	
Editing ex-engineers	▲	
Post-release failures		

that median values of the corresponding metric for FP components are statistically significantly lower than those for NFP components. Likewise, the symbol ▲ indicates that medians for the metric are significantly higher for FP components than for NFP components. Where neither ▽ or ▲ is plotted, no statistical difference between the two groups was observed.

A. *Microsoft Vista*

Table II presents the results from our investigation of VISTA. We noted that the sizes of components with outgoing dependencies for FP components are relatively smaller, however no such difference was found for incoming dependencies. Also number of methods in the classes was higher in the outgoing neighborhood of FP components. In the survey responses, only few developers linked size of depending components and failure-proneness.

The survey responses also indicated that most developers consider high degree of churn to increase failure-proneness. Our results supported their viewpoint in that we noted churn frequency for both incoming and outgoing dependencies to be higher for FP components, but in different directions. Churn frequency was lower for outgoing dependencies and lower for incoming ones. Size of churn was also lower for incoming dependencies. These observations suggest that when changing a component, it's vital to review all depending components as part of completing the task.

Code metrics such as fan in, fan out, and cyclomatic complexity are also lower in the outgoing neighborhoods of FP components. This suggests that depending on complex components is not necessarily risky, while depending on simple components is not necessarily safe either. Also noteworthy is that the dependees of FP components tend to have larger values for number of class methods, inheritance depth and class coupling. This suggests that the complexity of the API rather than code complexity increases the chances of having failures. Many of these results are opposite to the responses in the survey such as most developers considered high complexity and method-calls to increase failure-proneness. Also, all significant differences were noted only for outgoing dependencies.

To our surprise, we observed no significant differences for coverage and organization metrics. It appears that the likelihood of a component to fail is not related to how well its neighborhood is tested, possibly because failures are unlikely to propagate across dependencies and they remain local to the FP component — also, quite the opposite to the survey responses.

People measures seem to matter only for incoming dependencies. All three metrics were significantly higher for FP components and support findings in another study in which organizational structure [9] was found be to a strong predictor of the failure-proneness of a component. However, this seems to not hold only for incoming dependencies. Only a little over half the number of developers who responded in the survey considered people measures to be influential in failure-proneness.

Most surprisingly, we found that the number of past failures in the neighborhood of a component bears no relationship on its failure-proneness. Recall that nearly all developers from the survey believed such a link to be present!

It is noteworthy that only people measures were significantly different for incoming dependencies as compared to outgoing dependencies where we found more number of differences. Also, only one metric is common between both neighborhoods: churn frequency, and that too in opposite directions. These differences suggest treating the two types of dependencies separately and with caution. Combining the two metrics will likely mislead interpretations.

B. *Eclipse*

Our results from comparing the distributions of the medians of the metrics across FP and NFP components for ECLIPSE versions 2.0, 2.1, and 3.0 are presented in Table III. Size related metrics of incoming neighboring components have

Table III

RESULTS FROM ECLIPSE INDICATING SIGNIFICANT DIFFERENCES IN THE DISTRIBUTION OF METRICS ACROSS FP AND NFP COMPONENTS FOR INCOMING DEPENDENCIES. COMPONENTS.

		Incoming medians			Outgoing medians		
Category	Metric	E 2.0*	E 2.1*	E 3.0*	E 2.0	E 2.1	E 3.0
Size	Method lines of code (max)	▲			▲		▲
	Method lines of code (total)	▲			▲		▲
	Total lines of code	▲			▲		▲
Churn	Churn (added lines)	▲	▲	▲	▲		▲
	Churn (deleted lines)	▲	▲	▲	▲		
	Churn (changed lines)	▲		▲	▲	▲	▲
	Churned (total lines)	▲	▲	▲	▲		▲
	No. of commits	▲	▲	▲	▲		▲
Code metrics	Fan Out (max)	▲			▲		▲
	Fan Out (total)	▲			▲		▲
	Nested block depth (max)	▲		▲	▲		▲
	Nested block depth (total)	▲		▲	▲		▲
	Cycolmatic Complexity (max)	▲			▲		▲
	Cycolmatic Complexity (total)	▲			▲		▲
	No. of fields (max)						
	No. of fields (total)	▽					
	No. of interfaces				▽	▽	▽
	No. of methods (max)				▲		
	No. of methods (total)				▲		
	No. of classes				▲	▲	▲
	No. of static fields (max)	▽	▽				▲
	No. of static fields (total)	▽	▽				▲
	No. of static methods (max)						▲
	No. of static methods (total)						▲
	No. of parameters (max)	▲			▲		▲
	No. of parameters (total)	▲			▲		▲
Post-release failures	Count of Post-release failures	▲	▲	▲	NA	▲	▲

* Note that E 2.0 stands for ECLIPSE 2.0; E 2.1 for ECLIPSE 2.1; and E 3.0 for ECLIPSE 3.0.

significantly higher values for FP components for version 2.0, but not for other versions. Whereas the metrics are also significant for the outgoing neighborhood for versions 2.0 and 2.1. This means that larger depending components tend to increase the failure proneness of a component, which could be an effect of the open-source development model where several developers are involved and not all changes are strictly coordinated by a central group.

Churn metrics of neighboring components also seem to have higher values for FP components, i.e., larger number of changes in components may introduce errors that may propagate to depending components. This could also be an effect of incomplete changes in that depending components were not updated to sync with the intended changes. Interestingly, values for both size and churn metrics were lower for FP components in VISTA, but are higher in ECLIPSE.

The distributions of code metrics of neighboring components vary a lot across the FP and NFP components. For instance, fan out, nested block, and complexity measures are significantly higher for incoming dependencies in version 2.0 and outgoing dependencies in versions 2.0 and 3.0. Hence, higher complexity of the neighboring components for ECLIPSE tends to increase the failure-proneness of components, as opposed to VISTA in which we noted the very opposite. We suspect that the underlying reason for this difference lie in the development process models of the two projects. Having a large number of interfaces in outgoing dependencies also seems to be not an issue in the ECLIPSE project — to our surprise, the medians of the neighboring components have fewer interfaces for FP components, which means that using a larger number of services or data from other components is not risky. In the case of the incoming neighborhood, we observed no differences at all. In case of number of methods and classes, the outgoing neighborhood of FP components has larger medians meaning that the depending components undertake relatively a larger number of functions. Several other metrics such as static fields, static methods, and number of parameters were found to be significantly different, but the varied across versions (and also in direction in the case of static fields).

Lastly, past number of failures of neighboring components were signficantly higher for FP than NFP components for all our comparisons except for outgoing dependencies in ECLIPSE 2.0.

The results show that there is some discrepancy in the results between the different versions of ECLIPSE; results for ECLIPSE 2.1 were more markedly different than the other versions. A likely explanation for this is the nature of changes that may have been made to release ECLIPSE 2.1 in that changes may have less substantial in number of new functionalities or files added in comparison to ECLIPSE 2.0 and 3.0 (likewise, it is possible that there may be differences in the results when comparing different versions of Windows). There is also a greater degree of overlap between the metrics found to significantly differ across incoming and outgoing dependencies for ECLIPSE, while the overlap for VISTA was at best marginal. The results also differ from that of VISTA in that the metrics found to be significantly different in their distributions were often lower for FP components in VISTA and higher for ECLIPSE. Interestingly, the results from ECLIPSE appeared more compatible with the responses from the survey in comparison to VISTA.

VI. Related Work

In this section we discuss research that has used dependency metrics in empirical studies related to quality. Most prior work in this area has been done in the area of programming languages. Podgurski and Clarke [11] proposed a formal model for program dependences to evaluate several dependence-based software testing, debugging, and maintenance techniques. They also propose the notion of semantic dependence that models the ability of a program statement to affect the execution behavior of other statements. This notion is very similar in spirit to our underlying idea of the ability of dependencies to influence related components. Significant research has also focused on the generation of program dependence graphs [12]. Harrold et al. [13] have proposed techniques to generate program dependence graphs that contain the programs control flow as the program is being parsed .

Such program dependence information has been used to optimize and improve testing [14], debugging and maintenance [3], [11]. Rothermel and Harrold [3], for regression test selection, construct control flow graphs for a procedure or program and its modified version to use these graphs to select regression tests that execute changed code from the original test suite. Bates and Horwitz [14] propose the use of program dependence graphs with test adequacy data to reduce the time required to create new test files, and to avoid unproductive retesting of unaffected components. Dependences have also been used extensively in the program slicing [15], [16].

Dependencies have also been used in the STC to understand software development. Cataldo et al. [4] argue the importance of logical dependencies rather than call and data dependencies that have less impact on coordination requirements. Field studies performed by Cleidson de Souza et al. also indicate the importance of dependencies in coordination of work between teams of engineers [17].

Importantly, dependencies have been used for defect prediction. Schröter et al. [2] showed that import dependencies can be used to predict defects. Rather than looking at the complexity of a class, they looked exclusively at the components that a class uses. For Eclipse, the open source IDE they found that using compiler packages results in a significantly higher failure-proneness (71%) than using GUI packages (14%). Prior work at Microsoft [18]–[20] at Microsoft and more specifically on the Windows Server 2003 system [18] illustrates that code dependencies can be used to successfully identify failure-prone binaries with precision and recall values of around 73% and 75%, respectively. Shin et al. [21] observed that adding the calling structure information provided a marginal improvement (0.6%) increase in prediction accuracy compared to models based on non-calling structure code and historical attributes. Most prior research use dependencies to answer interesting research questions in the context of program analysis, testing, slicing, debugging and in the socio-technical area. None of the studies look at what characteristics in (or amongst) dependencies cause problems. To the best of our knowledge this is a problem that is still an open question that we hope our study forms the first step towards addressing. Also the prior work on using dependencies [2], [18], [19], [21] for prediction uses the dependency metrics as a metric in a larger set of predictors (with churn, complexity etc.) to predict failure-proneness. Our study involves leveraging the relationship (dependents, dependees) between the various code artifacts in analyzing the implications of dependencies. Additionally, prior research does not consider the characteristics of the component dependencies as is done in the study discussed in this paper.

VII. Threats to Validity

- **External validity:** The results of this study are from two software systems, both very large with many components, but also different in their development languages and model. 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 [22]. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment and projects for which it was conducted. While overall the results showed several commonalities across the two projects, generalizing them for smaller projects will require further investigation.
- **Internal validity:** In our study internal validity issues primarily deal with the experimental bias of our results. These concerns are addressed to some extent due to

the fact that the engineers at Microsoft had no knowledge that this study was being performed for them to artificially modify their behavior/coding practices or dependencies and the analysis was post-hoc (after the release of VISTA) to affect our results.

- **Construct validity:** Construct validity issues arise when there are errors in measurement. These issues are also alleviated to some extent by using the median values in our analysis, and by the large size and diversity of our dataset. Also, the components identified as non-failure prone may have defects that have never surfaced and therefore are not possible to account for.

VIII. Conclusions and Consequences

In this paper, we report an empirical study that analyzes the relationship between post-release failures and characteristics of incoming and outgoing call and data dependencies among components. The specific characteristics analyzed were size, churn, code complexity, code coverage, organization information, and failure-proneness. Our empirical results from the two investigated projects in some cases confirm often believed assumptions about dependent pieces of code and in many other cases, they are counterintuitive.

Tables II and III present the results of our experiments on the two projects. The intention behind using projects with marked differences was to be able to reflect on the generality of the findings. We found several commonalities in the results, but also some differences. For instance, contrary to common belief, we found that frequency of change and complexity of components reduces failure proneness of their dependencies in VISTA, while on the other hand these metrics were found to increase risk in ECLIPSE. Further in-depth investigation into these projects will help us understand the causes for these differences, which we plan to accomplish in future work.

Some of the noteworthy counter-intuitive results, as in divergent from the survey results are:

- Depending on a component that has failures does not have an effect on failures of the dependent component (in VISTA).
- Depending on a binary with higher coverage (implying more testing) does not have an effect on failures of the dependent component (in VISTA).
- Being dependent on a component that has more engineers (or ex-engineers) contributing code to it does not relate to the number of failures. Being a dependee has an opposite and adverse relationship (in VISTA).
- It appears that the complexity of the API rather than code complexity increases the chances of having failures (for VISTA).
- Being dependent on components with fewer interfaces increases the risk of failure proneness (in ECLIPSE).
- The responses from the survey on Microsoft developers were more in sync with the results from ECLIPSE than with VISTA.

Other surprising observations include size, churn metrics, and code metrics like fan in, fan out, and cyclomatic complexity are related to fewer failures; but in the case of ECLIPSE, we observed that such metrics are indicative of more failures. To some degree most prior work [23]–[25], including work at Microsoft [26] has focused primarily on looking at the relationship between code metrics and failures. This contrast between VISTA and ECLIPSE raises questions regarding which properties of the projects determine or influence the relationship between the dependencies and software quality.

We expect that many of the observations made from our work is insightful and can be used to assist design and develop software and aid engineers in assessing the risk of forming new or supporting existing dependencies. We also believe that our results can be leveraged in the defect prediction [2] and test prioritization [3] domains to provide an additional dependency factor in the analysis of large systems. Additionally our results are useful to provide empirical evidence on an important problem in software engineering: the impact of dependencies on failures. Our results also provide additional empirical evidence on the importance of dependencies to researchers in the socio-technical congruence [4] domain.

Our future work plans have two main avenues: first is to collaborate with other researchers and replicate the study in more domains and contexts, both in the open source and closed source communities. As this is the first study in this area we will collect such results to aggregate them to build an empirical body of knowledge to aid engineers. Secondly we plan to leverage this data to collaborate with visualization researchers to build tools that will add context and provide domain knowledge to engineers working on these systems to understand the associated risks with the projects. We plan to conduct an empirical evaluation on the efficacy of such a visualization tool on the software development environment.

Acknowledgements

Laurie Williams was a visiting researcher in Microsoft Research (MSR) when this work was performed. We would like to thank the Windows team at MSR and Brendan Murphy of Microsoft Research, Cambridge for his help in understanding the data sources. We also thank the anonymous reviewers for their feedback on previous versions of this paper.

References

[1] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engg.* Addison-Wesley, 1995.

[2] A. Schröter, T. Zimmermann, and A. Zeller, “Predicting component failures at design time,” in *ISESE ’06: Int. Symposium on Empirical Software Engg.* New York: ACM, 2006, pp. 18–27.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *TOESM*, vol. 6, no. 2, pp. 173–210, 1997.

[4] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity," in *ESEM '08: Int. Symposium on Empirical Software Engg. and Measurement*. New York: ACM, 2008, pp. 2–11.

[5] A. Srivastava, J. Thiagarajan, and C. Schertz, "Efficient integration testing using dependency analysis," Microsoft Research, TechReport MSR-TR-2005-94, July 2005.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *PROMISE '07: Workshop on Predictor Models in Software Engg.*, Minneapolis, USA, May 2007.

[7] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Int. Conf. on Software Engg.* New York: ACM, 2005, pp. 284–292.

[8] T. J. McCabe, "A complexity measure," in *ICSE '76: Int. Conf. on Software Engg.* Los Alamitos, CA, USA: IEEE Press, 1976, p. 407.

[9] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *ICSE '08: Int. Conf. on Software Engg.* ACM, 2008, pp. 521–530.

[10] J. Shaffer, "Multiple hypothesis testing," *Annu. Rev. Psychol.*, vol. 46, no. 1, pp. 561–584, 1995, an expanded version appeared as Multiple hypothesis testing: A review. National Institute of Statistical Sciences Technical Report No. 23, September, 1994.

[11] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE TSE*, vol. 16, no. 9, pp. 965–979, 1990.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[13] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," *SIGSOFT Software Engg. Notes*, vol. 18, no. 3, pp. 160–170, 1993.

[14] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *POPL '93: Symposium on Principles of Programming Languages*. New York: ACM, 1993, pp. 384–396.

[15] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *PLDI '88: Conf. on Programming Language Design and Implementation*. New York: ACM, 1988, pp. 35–46.

[16] M. Weiser, "Program slicing," in *ICSE '81: Int. Conf. on Software Engg.* Piscataway NJ: IEEE Press, 1981, pp. 439–449.

[17] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: the multiple roles of apis in software development," in *SIGSOFT '04/FSE-12: Int. Symposium on Foundations of Software Engg.* New York: ACM, 2004, pp. 221–230.

[18] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM '07: Int. Symposium on Empirical Software Engg. and Measurement*. Washington DC: IEEE, 2007, pp. 364–373.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE '08: Int. Conf. on Software Engg.* ACM, May 2008, pp. 531–540.

[20] ——, "Predicting subsystem failures using dependency graph complexities," in *ISSRE '07: IEEE Int. Symposium on Software Reliability*. Washington DC: IEEE, 2007, pp. 227–236.

[21] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, "Does calling structure information improve the accuracy of fault prediction?" *Mining Software Repositories, Int. Workshop on*, vol. 0, pp. 61–70, 2009.

[22] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE TSE*, vol. 25, no. 4, pp. 456–473, 1999.

[23] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE TSE*, vol. 22, no. 10, pp. 751–761, October 1996.

[24] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE TSE*, vol. 31, no. 10, pp. 897–910, 2005.

[25] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04: Int. Symposium on Software Testing and Analysis*. Boston: ACM, July 2004, pp. 86–96.

[26] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," in *ISSRE '06: Int. Symposium on Software Reliability Engg.* Washington DC: IEEE, 2006, pp. 62–74.