The Impact of Interface Complexity on Failures: an Empirical Analysis and Implications for Tool Design

Marcelo Cataldo[1], Cleidson R. B. de Souza[2],
David L. Bentolila[2],Tales C. Miranda[2], Sangeeth Nambiar[3]

January 2010
CMU-ISR-10-100

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

[1] Institute for Software Research, School of Computer Science, Carnegie Mellon University
[2] Department of Informatics, Universidade Federal do Para
[3] Robert Bosch Engineering and Business Solutions Ltd.

ABSTRACT

Information hiding is a cornerstone principle of modern software engineering. Interfaces are central to realizing the benefits of information hiding, but despite their widespread use, designing good interfaces is not a trivial activity. Particular design choices can have a significant detrimental effect on quality or development productivity. In this paper, we examine the relative impact of interface complexity on the failure proneness of source code files using data from two large-scale systems from two distinct software companies. Our analyses showed that increases in the complexity of interfaces are associated with increases the failure proneness of source code files. Building on our empirical results, we develop a set of implications for designing tools aimed at assisting developers to cope with the detrimental impact of interface complexity.

Keywords: APIs, API design, empirical software engineering, software quality, development tools.

1. INTRODUCTION

Modularization is the predominant design approach for complex engineering systems [1, 13, 45, 48]. Modular systems, the theory argues, are superior to other designs because they minimize the costs of change, evolution, testing and experimentation [1, 52]. In the context of software engineering, the concept of information hiding proposed by Parnas [45] is a fundamental principle that allows software architects and designers develop modular software systems. This concept has been instantiated as data encapsulation, interfaces, polymorphism and other ways in modern programming languages [37]. Interfaces, in particular, are widely regarded as “the only scalable way to build systems from semi-independent components” [27]. They allow software developers to work in parallel and minimize the impact of their colleagues’ work [18, 45]. Despite their widespread use and acceptance in the industry [19, 20], designing good interfaces is not a trivial activity [22, 30]. There are different guidelines for an API implementation [8, 20], which are solely based on each author’s experience. In addition, past work has examined how design choices in terms of interface design impact quality attributes such as maintainability [4] and usability [22], as well as tool development to assist on the usage of interfaces [15, 17, 51, 57].

However, limited attention has been given to systematically evaluating the impact of design attributes of interfaces such as their complexity and ease of use on traditional outcomes such as software quality and development productivity. Recent work has examined the role of interface usability [e.g. 22] on software development projects. However, the impact of interface complexity has been, mostly, neglected. This lack of research on the relationship between “good” API design and its impact on software projects is a gap that needs to be addressed because it represents a potential barrier for development organizations towards fully realizing the benefits of modular systems.

In this paper, we examine the relative impact of interface complexity on the failure proneness of source code files using data from two large-scale systems from two distinct software companies. We utilize interface complexity measures proposed by Bandi and colleagues [4]. Our empirical analyses showed that increases in the complexity of interfaces are associated with increases in the likelihood of source code files being part of a post-release defect. We, then, use these results to guide the development of a set of design implications for tools that can assist software developers cope with the impact of interface complexity.

The rest of the document is organized as follows. First, we discuss the research background that motivates the research questions examined in this paper. Second, we present our empirical analysis followed by a description of an example of how visualization tools can assist developers. Finally, we discuss limitations and implications for future research.

2. THE ROLE OF INTERFACE COMPLEXITY ON SOFTWARE QUALITY

Design decisions have an important impact on the ability of software systems to achieve their functional and non-functional requirements [7, 10, 53]. Principal decisions articulated at the architectural level serve as a general framework that guide and constrain lower level and more detailed design decisions [53]. Two critical and interrelated attributes of a software system are impacted by design decisions: the allocation of functional responsibilities within specific constituent parts of a system (e.g. modules or components) and relationships among those parts. Those relationships are typically realized in the form of interfaces which play a very important role in the development process of software system from a technical point of view as well as from an organizational perspective.

In the technical dimension, interfaces determine a host of attributes of a system such as the efficiency of the communication between modules or components [e.g. 10], the easiness and efficiency of accessing the functionality of a module or a component [e.g. 34], the usability and understanding of the interfaces [e.g. 22], as well as the evolution and maintainability of the system [10, 30]. Then, poorly designed interfaces could represent an important technical liability for software systems, particularly, for those large-scale complex systems that are pervasive in today's world.

On the organizational side, the design of a module or a component interface defines a key set of coordination needs for a development organization [12, 32, 49]. Unsatisfied coordination needs tend to results in misunderstandings and mistake which, typically, manifest themselves as higher levels of defects [12, 14, 16, 33]. Moreover, uncertainty around an interface creates a number of coordination requirements that tend to be difficult to identify [16, 27].

Unfortunately, designing good interfaces in technical and organizational terms is not a trivial task [30]. Different guidelines have been proposed for that [8, 20, 57]. However, those guidelines are based on the experiences of a subset of practitioners and the work lacks a systematic empirical evaluation of the proposed guidelines. More importantly, only a limited set of studies have investigated how the design of interfaces relates to quality attributes of a system like usability and maintainability [e.g. 22]. In particular, an aspect of interface design that has been mostly neglected by researchers is design complexity. One traditional view of complexity focuses on properties of the implementation of a particular function or methods and an extensive literature has developed since Halstead and McCabe seminal work on code complexity [28, 39]. Another influential view of design complexity relates to structural properties of a software system and the interconnections among constituent parts. The concepts of coupling and cohesion [50] are central in this line of work. The dimension of design complexity of interest in the context of interfaces represent a third line of work that has received limited attention, particularly, in the context of empirical analysis of measures and their impact on traditional software engineering outcome variables such as quality and development productivity. One such example is the work of Bandi and colleagues [4] who examined the impact of design complexity of interface specifications to maintenance tasks of software systems. The authors used a set of metrics (e.g. interface size and operation argument complexity) to assess the complexity of interfaces. These metrics are based on the types and the number of parameters a method or a function has. They are somewhat intuitive: they rely on the assumption that a method or function with a large number of parameters and whose parameters are objects is said to be more complex than another method or function with fewer parameters based on primitive types (e.g. integers). Bandi and colleagues [4] found that maintenance tasks with higher levels of interface complexity took longer to resolve than those that involved less complex interfaces. Their work provides empirical evidence that interface complexity is detrimental to development productivity. On the other hand, the relationship of interface complexity with software quality has been neglected. This paper addresses this gap in the literature by examining the following research question:

*RQ1: **What is the relative impact of interface complexity on failure proneness?***

Despite the lack of systematic empirical examination of the impact of design attributes of interfaces such as complexity on software quality, recent work has focused on the development of tools to aid in the usage of interfaces and APIs[1]. Examples of such tools are mashups editors, examples-based interface learning support, as well as directives for providing context-based information for interface users [15, 51, 57]. Researchers have also built on qualitative analyses of developers' usage of APIs to suggest design implications for development tools [16, 28]. Robillard [47] identified several problems Microsoft software developers faced when learning new APIs. His results can be used to suggest design implica-

[1] Loosely speaking, an interface is the set of public functionalities exported by a software component, e.g. public methods of a class. An API, on the other hand, is the combination of several interfaces. For instance, a Java API consists of several public classes and their methods. Our analysis described in section 3 focuses on interfaces, while our discussion on section 4 addresses both interfaces and APIs, since research tools have focused more on APIs than on interfaces.

tions for API support tools. Despite those important advances – with significant value for researchers and practitioners – an important next step is to be able to provide recommendations for tools to support the design of APIs. Building on our empirical examination of the relationship between interface complexity and software quality, we suggest a collection of design recommendations for tools. Then, our paper has two main contributions to the software engineering literature. Addressing research question 1 is our first contribution, while the set of recommendations for tool design represents our second contribution.

3. EMPIRICAL STUDY

In this section, we described our empirical analysis of the impact of interface complexity on software failures. We collected data from two large software development projects from two distinct companies. The first project was a complex distributed system produced by a company operating in the computer storage industry, while the second was an embedded system from a company that operates in the automotive industry. In the remaining of this paper, we refer to the first project as "System A" and the second project as "System B".

3.1 Research Settings

The company developing System A had one hundred and fourteen developers grouped into eight development teams distributed across three development locations. All the developers worked full time on the project during the time period covered by our data. The data covered a period of 11 months of development activities corresponding to the latest release of the company's product. Those development activities were captured by 1125 modification requests. The modification requests represented different types of development activities from implementation of new features to fixing defects. The system was composed of approximately 5 million lines of code with most of the code written in C and a relatively small fraction in C++. The APIs exported and used by the various source code files were primarily of two types: traditional function call that interfaces layers of the system that run in the same runtime entity and remote procedure calls that were used exclusively for interfacing among runtime entities operating in the same or different computational node.

System B and the development organization that produced it had very different characteristics compared to the context of System A. The development organization used a product line approach. Our data covered 4 years of development activities in the latest version of the base platform software grouped in 3,840 modification requests. Three hundred and eighty developers distributed in eight locations across Europe and Asia participated in the development. The system was composed of approximately 7 million lines of code organized in 530 architectural components. All source code files were written in C language. A collection of XML files were used to define data elements and interfaces. The compilation process used such XML files to auto-generate the data structure and interface declarations that the C source code files defined and used. In System B, the interfaces were a combination of function calls and global variables used for passing data among interdependent files or architectural components.

3.2 Description of the Measures

The basis of our data collection was the development activity represented by a modification request. In both development organizations, every change to the source code was controlled by modification requests. A modification request (MR) is a development task that represents a conceptual change to the software that involves modifications to one or more source code files by one or more developers [40]. The changes could represent the development of new functionality or the resolution of a defect encountered by a developer, the quality assurance organization, or reported by a customer. Then, the version control system and the MR-tracking systems of the development organizations constituted the main sources of data. In addition, we utilized the source code itself as a third important source of data. Combining all three data sources, we constructed our measures.

The selection of outcome and independent measures was based on the work by Cataldo and colleagues [12] that examined the failure proneness of source code files. Such analysis combined traditional metrics (e.g. code size and churn metrics) with structural properties of the dependencies of the source code files (e.g. syntactic and logical dependencies). The examination of the role of API complexity represents a natural extension to such work because it combines the structural properties of dependencies with an additional dimension, the *complexity* of those linkages among source code files. The measures are described in the subsequent paragraphs.

3.2.1 Measuring Failure

Our outcome variable of interest is failure proneness at the source code file level. Our dependent variable, *File Buggyness*, was measured as a dichotomous variable indicating whether a file has been modified in the course of resolving a "field" defect. In the case of System A, field defects represent instances of problems reported by customers after the product has been released. In the case of System B, we consider "field" defects as those encountered in the integration and system testing phase of the development process. A field defect in the industry setting of System B would result in product recall and significant financial impact. Then, an important amount of effort is spent in the system testing phase to make sure that defects are found before the product is released to customers. Therefore, field defects seldom occur. Given the binary nature of our outcome variable, a logistic regression model was used to examine the role of API complexity on software failure proneness.

3.2.2 Measuring API Complexity

We constructed several measures of API complexity using metrics proposed by Bandi and colleagues [4]. For each source code file, we first identified the set of interfaces those files made accessible for other source code files (e.g. public methods and variables for C++ code and *extern*-ed functions and variables for the C portion of the code). Then for each interface, we computed the interface size and operation argument complexity proposed as described by Bandi and colleagues [4]. Interface size is defined as the number of parameter plus the sum of the parameters' type sizes and both terms are multiplied by constants. In our computations, we set those constants to 1 (see [4] for a discussion of selecting the constant values). The operation argument complexity measure is the sum of the parameters' type sizes. Bandi and colleagues [4] did not consider the case where interfaces are data elements such as global variables. In those cases, we can define the interface size and operation argument complexity measures in different ways. One possibility is to assume that they are equivalent to a "parameterless" interface and assign a 0 to both measures. Alternatively, we could assign to both measures the value corresponding to the data element type as if it were a regular interface with a single parameter. We evaluated both approaches and the results were similar. In section 3.4, we report the results based on the second approach. Finally, the interface-level measures were aggregated at the level of the source code file. For each file, we calculated a total sum measure of interface size and operation argument complexity as well as average, maximum, minimum and standard deviation variants of the measures.

3.2.3 Additional Factors Impacting Software Quality

Past research work has examined the role of numerous types of metrics on software failures [21, 23, 26, 41, 42, 48]. Process-oriented measures such as number of changes, number of deltas, and age of the code have been shown to be very good predictors of failures [26, 41]. Therefore, we collected *Number of MRs*, which is the number of times the file was changed as part of a past defect or feature development and the *Average Number of Lines Changed* in a file as part of past modification requests. Although, product-related measures such as code size and code complexity measures have produced somewhat inconsistent results as predictors of software failures [6, 9, 26], senior engineers from the development organizations we studied indicated that in their experience product measures had a positive relationship with software defects. We measured size of the file as the number of non-blank non-comment lines of code.

We also collected the syntactic and logical dependencies metrics examined by Cataldo and colleagues [12]. Each one of these metrics is discussed below. Syntactic dependency information was collected in System A using a modified version of the C-REX tool [29] to identify programming language tokens and references in each entity of each source code file. In the case of System B, the data was collected using an existing tool that the organization had deployed for performing various types of code level analytics. Snapshots of the source code corresponding to the last release of both systems were used to collect the syntactic dependency information associated with data, function and method references crossing the boundary of each source code file. In this paper, we refer to data references as data dependencies and function/method references as functional dependencies. We consider that differentiating the dependencies across types is important because the nature of their impact differs as argued by Cataldo and colleagues [12]. We computed the number of inflow and outflow for both data and functional syntactic dependencies.

Logical dependencies, based on the idea of logical coupling proposed by Gall and colleagues [25], relate source code files that are modified together as part of an MR. If the modification request is implemented by making changes to only one file, such instance of a development activity does not provide evidence of any dependency between the modified file and the rest of the system. On the other hand, when the modification request requires changes to more than one file, decisions about the change to one file suggest the existence of some time of dependency with decisions made about changes to the other files involved in the MR. As discussed earlier in this section, both organizations required that any change to the system be associated with a modification request. Using such data, we constructed a logical dependency matrix among the source code files. Unlike the syntactic dependency information which has a clearly defined directionality (a callee and a caller), the logical dependency matrix is a symmetric matrix, it is undirected. Each cell C_{ij} in the matrix represents the sum of the number of times files *i* and *j* were changed together as part of an MR that were resolved over the time period covered by our data.

The quality of the logical dependency data relies on the adherence of developers' actions to the defined change submission processes. For instance, a developer could submit a commit containing changes to two different files but those changes are associated with different modification requests and they do not related to an actual dependency among the files. A collection of analyses were performed to assess the quality of our MR-related data and minimize measurement error. We compared the revisions of the changes associated with the modification requests and we did not find evidence of such type of behavior. One of the authors together with a senior engineer of each organization examined a random sample of modification requests to determine if developers have work patterns that could impact the quality of our data such as the example described above. We did not find commits in the version control systems that contained modifications to the systems' code that was unrelated to the development task represented by the modification requests. Two file-level measures were extracted from the logical dependency matrix, the number of logical dependencies and the clustering of logical dependencies. Unlike the number of logical dependencies, the clustering of logical dependencies measure captures the degree to which the files that have logical dependencies to the focal file have logical interdependencies among themselves. In graph-theoretic terms, the measure for file *i* is computed as the density of connections among the direct neighbors of file *i*, a definition equivalent to Watts's [55] local clustering measure. We refer the reader to Cataldo et al [12] for the formal details of the measures.

3.3 Preliminary Analyses

As has been discussed in past research [e.g. 12, 41, 42], many of the churn metrics and source code measures tend to be correlated. Therefore, we started our analyses with a collinearity diagnostics. We used a variance inflation factors analysis to identify those independent variables that were highly correlated and, consequently, impact the quality of the estimates of our regression models. The results of the collinearity diagnostics indicated that several measures were highly correlated. The number of modification requests measure was correlated with several other variables. The syntactic dependency measures were also highly correlated among themselves. The interface size measure was highly correlated with the operational argument complexity measures. The final set of independent variables included in our analyses is indicated in table 1 and consists of the following measures: size in LOCs, average change in LOCs, number of inflow syntactic data dependencies, number of inflow syntactic functional dependencies, number of logical dependencies, clustering of logical dependencies, sum of interface sizes and standard deviation (dispersion) of interface sizes. We calculated the pair-wise correlations among those measures and the highest one were between size of the file in LOC and the number of logical dependencies and the sum of interface sizes, 0.348 and 0.312, respectively. Descriptive statistics showed that several variables included in the models were highly skewed, therefore, we log-transformed them.

We followed a standard hierarchical modeling approach. We constructed a baseline model for both systems consisting of the factors impacting failure proneness identified by Cataldo and colleagues (see models I and III in table 1). We, then, introduce the API complexity measures in models II and IV in table 1. We report the Chi^2 of each model, the percentage of deviance explained by each model as well as the statistical significance of the difference between a model that adds new factors and the previous model without the new measures. The percentage of the deviance explained (where deviance is defined as -2 times the log-likelihood of the model) is a ratio of the deviance of the null model (containing only the intercept), and the deviance of the final model. For each model, we report the odds ratios associated with each measure instead of regression coefficients. Odds rations provide a simpler way of interpreting the impact of a particular factor on the outcome variable. For instance, an odds ratio of 1.5 for a dichotomous independent variable indicates that a change from 0 to 1 increases the likelihood of positive changes in the outcome variable by 50%. Odds ratios larger than 1 indicate a positive relationship between the independent and dependent variables whereas an odds ratio less than 1 indicates a negative relationship.

Table 1: Odd Ratios from Logistic Regressions

	System A		**System B**	
	Model I	**Model II**	**Model III**	**Model IV**
Size in LOCs (*log*)	1.206**	1.313**	1.258**	1.242**
Avg. Change in LOCs (*log*)	1.146**	1.019*	1.315**	1.321**
No. Inflow Syntactic Data Dependencies (*log*)	1.047	1.042	0.993	0.997
No. Inflow Syntactic Functional Dependencies (*log*)	0.976	0.912	1.034	1.029
No. Logical Dependencies (*log*)	2.304**	2.109**	3.827**	1.892**
Clustering of Logical Dependencies (*log*)	0.007**	0.016**	0.002**	0.005**
Sum of Interface Sizes (*log*)		1.812**		1.427**
Dispersion of Interface Sizes (*log*)		1.002		1.041*
N	2802	2802	9074	9074
Model Chi^2 (p-value)	1195.59 ($p < 0.01$)	1372.94 ($p < 0.01$)	3098.27 ($p < 0.01$)	3116.81 ($p < 0.01$)
Deviance Explained	30.87%	35.43%	36.67%	37.02%
Model Comparison (p-value)	--	176.35 ($p < 0.01$)	--	18.54 ($p < 0.01$)

(* $p < 0.05$, ** $p < 0.01$)

3.4 Results

Table 1 reports the results of examination of the relative impact of interface complexity on failure proneness. The results associated with System A are reported in models I and II while models III and IV present those results associated with System B. The baselines models (I and III) included the set of factors reported in Cataldo et al [12]. We see that in both models, the measures size, average change and number of logical dependencies have odds ratios higher than 1 and are statistically significant, which indicates that higher values of the measures increase the likelihood of failure on the source code files. The analyses also show that files logically dependent on files which are also highly interdependent have lower likelihood of being associated with defects, as indicated by the odds ratio lower than 1 associated with the clustering of logical dependencies measure. All these results are consistent with those reported by Cataldo et al [12].

Models II and IV introduce our measures of interface complexity. We observe that the sum of interface sizes has an important and statistically significant impact on source code files failure proneness. In the case of system A (model II), the odds ratio associated with the measure has a value of 1.812 indicating that a unit increase in the log-transformed variable increases the likelihood of source code files of being associated with defect by 81.2%. The results are similar for the case of System B (model IV), although the magnitude of the impact of interface size is about half than the case of System A. One possible explanation for this difference is the nature of the interfaces use in both systems. System B made extensive use of global variables in System B. The interface size measure, as discussed earlier, might not necessarily fully capture the potential detrimental impact of the complexity associated with global variable. Then, the impact of the factor could be under estimated.

While the sum of interface sizes gives a general indication of the complexity associated with interacting with a particular file, the dispersion of interface sizes measure provides an indication of how such complexity is distributed across the interfaces exported by a particular file. Model II and IV show that the dispersion of interface sizes is only statistically significant in the case of system B. Increases in the dispersion of complexity are associated with higher levels of failures although the magnitude of the impact of this factor is relatively small compared of the other factors.

3.5 Further Exploration of the Results

We also performed additional exploratory analysis with visualizations. We extended the tool Metrix [17], which provides automatic evaluation of complexity of Java-based interfaces and APIs, to manage our data and relate interface complexity to number of defects and number of users of the interfaces. Figure 1 contains four quadrants which depict association patterns between interface complexity and defects (left quadrants) and number of users of the interfaces (right quadrants) for both of our systems. In each quadrant, the squares represent a file containing several interfaces. The size of the square in the left quadrants indicates the number of defects associated with the file, the larger the square, the more defects associated with that file. The color of the square represents the sum of the level of complexity of each interface in that file with red color shades indicating higher levels of interface complexity and green shades indicating lower complexity. The same definitions apply to the left quadrants but the focus is on number of callers instead of number of defects.

The figure depicts some interesting patterns. First, we observe that in System A (top two quadrants) complex interfaces tend to be associated with buggy files (red color of the squares in the top left) and tend to be used extensibly (red color of the squares in the top right). On the other hand, in System B, we see the opposite pattern. Highly complex interfaces that tend to be associated with buggy files (red color of the squares in the bottom left) tend to have low number of callers (green color of the squares in the bottom right). These patterns could suggest that there is a difference in the set of design decisions made by architects and designers of each system. Those decisions (e.g. highly complex interfaces that are heavily used) can have serious consequences on the quality of a software system. Certainly, the detrimental impact would be moderated by organizational factors such as experience and processes. However, we think this type of analysis which combines statistical evaluation of historical data with visualization of key data relationships has significant value for researchers as well as practitioners because they could uncover interesting patterns in the data based on solid statistical ground. We discuss these implications further in the discussion section.

Figure 1 also highlights some particular sets of files. For instance, buggy files with low levels of interface complexity (large size squares with color green) as well as files with high levels of interface complexity that tend to be associated with fairly small number of defects. We examined in more detail the attributes of these sets of particular files which happened to be relatively small in size (System A: 447.88 average LOCs, 378.5 standard deviation – System B: 339.56 average LOCs, 433.06 standard deviation). In the case of buggy files with low levels of interface complexity, we found that the only differences were related to the logical coupling of the files. This set of particular files had significantly higher levels of logical coupling (System A: $t = -4.85$, $p < 0.001$ – System B: $t = -10.06$, $p < 0.001$) than other files with similar size. In addition, those files have significantly lower levels of clustering in the logical dependencies compared to similarly sized files (System A: $t = 2.84$, $p < 0.001$ – System B: $t = 10.65$, $p < 0.001$). These results enforce the important impact of logical coupling on failure proneness as demonstrated by Cataldo and colleagues [12]. In the case of files with high interface complexity and low number of defects, we did not find any statistically significant differences with other files.

4. DISCUSSION

In this paper, we have examined the relative impact of interface complexity on the failure proneness of source code files. Our results showed that higher levels of interface complexity of source code files (measured as sum of the size of the file's exported interfaces) are associated with increases in the "bugginess" of files or likelihood of source code files being associated with a defect. Our work has several important contributions to the software engineering literature. First, we extended our knowledge about the role of interface complexity by examining Bandi and colleagues measures in the context of software failures. Our results are complementary to those of Bandi et al [4] since their results suggested that higher interface complexity was positively associated with maintenance time. Second, our analyses combined multiple factors that impact failure proneness. In particular, we extended traditional empirical analyses which focused on churn and product-related metrics with complexity and relational (in the form of syntactic and logical dependencies) characteristics of the interfaces. Finally, we replicated our results across two systems from two different companies strengthening the external validity of our results. The remainder of this section explores the implications of our results for tool support as well as further empirical.

4.1 Interface Complexity and Quality

Software complexity has been an important research topic for several decades [5] and the increasing pervasiveness of software systems suggests that the topic will remain relevant in the future. This line of work has traditionally focused on assessing the complexity of a particular unit of software such as functions, modules or components [e.g. 24, 38, 39, 56] and examining its impact in the context of software maintenance activities and software quality [5, 35, 24]. Numerous code complexity measures have been proposed (see [59] for a comprehensive list), however, empirical examinations of the relationship between complexity and quality have produced disappointing results [24]. Fenton and Ohlsson [24, page 808] argued that "…being a good predictor of fault density … is not an especially appropriate validation criteria … Since complexity infers the notion of difficulty in understanding … such metrics should be that they are good predictor of maintainability …". In fact, such observation has been confirmed empirically by work such as Banker et al [5].

Our study, on the other hand, examined a dimension of complexity – interface complexity - that has been relatively neglected. We are addressing an important gap in the literature because interfaces are a central element in modular systems. The modularity literature [e.g. 1, 52] argues that interfaces are the link between modules or components that allows for separation of concerns and development activities, therefore facilitating the coordination among developers. Such argument rests on the assumption that complexity is embedded in the software entity (e.g. module or component) and not in the interfaces themselves. Our work and other recent research [e.g. 4, 16, 18] suggest a departure from that line of work where complexity is in fact embedded in the interfaces themselves. Consequently, the separation of technical and work responsibilities is not as simple and pristine as suggested by the modular systems theoretical perspective. Then, interfaces have the potential to become barriers to effectively decoupling technical responsibilities and work responsibilities as well as hinder coordination among development teams. The work presented in this paper represents a first step towards characterizing the complexity of interfaces and empirically assessing its impact on software quality. The following paragraphs discuss several future research directions we consider will further our understanding of the relationship between interface complexity and software quality.

4.1.1 The Nature of Interface Complexity

The complexity associated with an interface relates to multiple dimensions. In this paper and building on prior work, we considered complexity as a function of the number and type of parameters of a particular interface. However, there are additional dimensions that future research should explore. First, the set of pre- and post-invocation made by the interface designers and implementers represent also important source of complexity and, consequently, a potential factor leading to failures [43, 46]. For instance, a developer of a module's interface has some system configurations for which he/she had developed and validated the particular piece of software. Documentation and communication of these configurations in such a way that it is able to capture all the relevant details is a challenge, leading to a variety of defects when those interfaces are utilized. Past research in static analysis [3] has proposed approaches to address problems related with misalignment of assumptions between the interface provider and user. However, such research focuses on basic issues such as verification of lock/unlock policies and conformance of particular code structures in device drives. Nambiar and Cataldo [43] presented a set of cases that related to a higher order of misalignment of assumptions that are associated with the information content exchanged through the interfaces as well as the environmental context in which the interface is expected to operate. Another, but related, dimension of complexity is related to the sequence of invocation of different interfaces and the negative consequences that incorrect or unanticipated sequences can have on software quality [34].

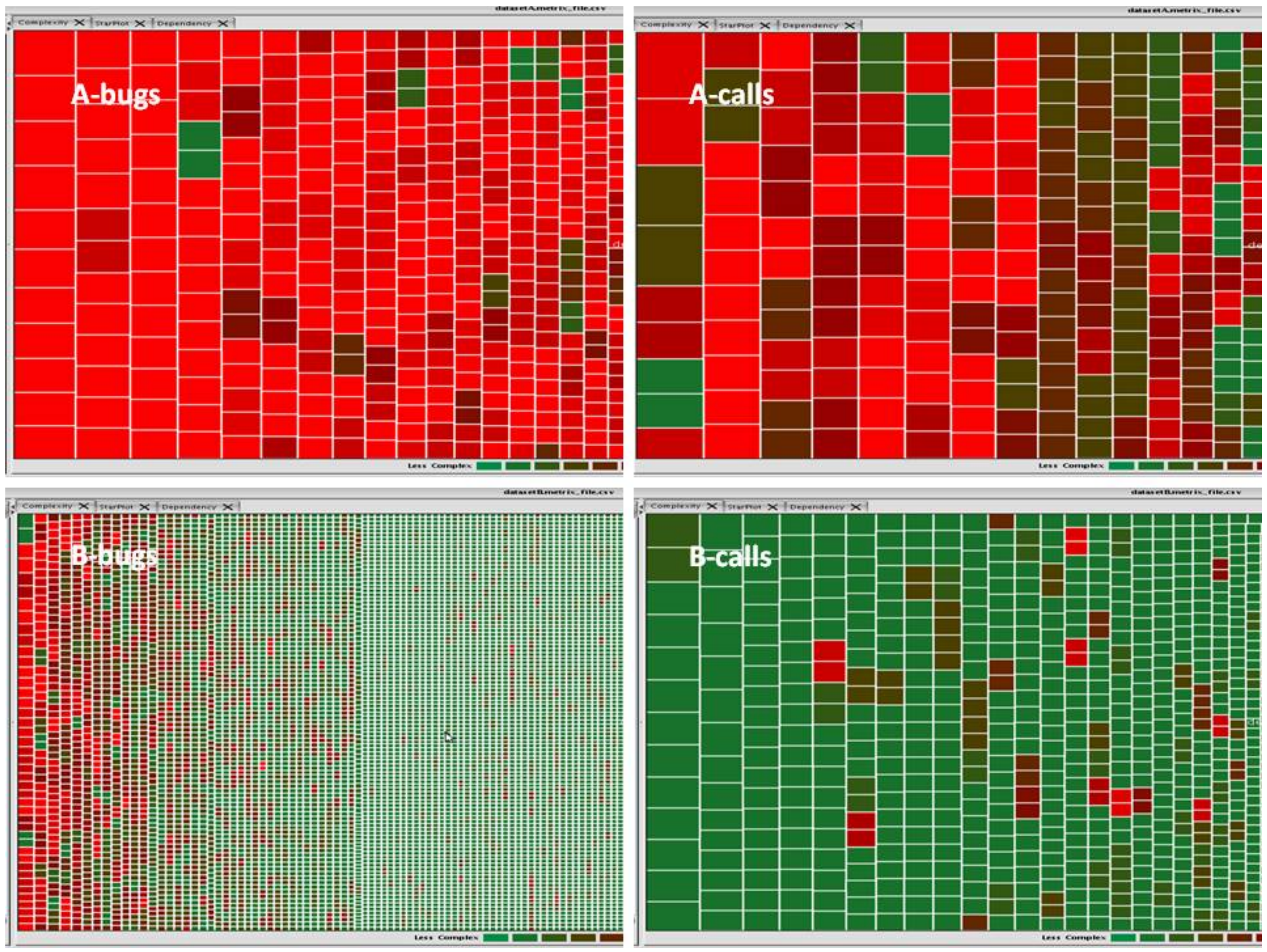

Figure 1: Pattern of Defects and Usage of Interfaces in Systems A and B.

Another area that deserves further examination is the implications of particular usage patterns of parameter types for interface complexity. For instance, a function or method may take a single integer parameter. In the context of Bandi et al's measures, such interface has an operational argument complexity of 1 (the size of the parameter type as defined by [4]). However, the function or method could use such parameter as a mechanism for controlling a complex sequence of statements (e.g. a mask). Arguably, this example implies more complexity in the use of the interface because it requires the function or method user to understand a lot more about the interface and potentially about the implementation and the implications of particular choices of bits in the mask. This is similar to what Kiczales has termed as "open implementation" [36]. Then, future research should explore ways to capture such differences in complexity into new metrics.

4.1.2 Coordination of Geographically Distributed Development Work

de Souza and Redmiles [18] found that dependencies associated with interfaces relating two or more development teams might not be easily identified by those developers. Lack of awareness or disregard for particular dependencies can have important consequences on the quality of a software system. Past research work has shown that unsatisfied coordination needs can result in coordination breakdowns, particularly in a geographically distributed setting [33, 44]. Those coordination breakdowns tend to materialize as higher number of defects [12, 33]. Our results showing the detrimental impact of interface complexity on software quality suggest an additional possible source of dependency among the users and the "supplier' of the interfaces that might be difficult also to identify and manage. Then, we could consider the complexity of APIs as a factor to help the design of distributed teams that would be better equipped to handle the dependencies imposed by the technical properties of the system. For instance, teams dealing with more complex APIs could be located closer (physically or in terms of time zone difference) than those teams dealing with simpler APIs. In other words, future research should explore the use of metrics that characterize the nature of interfaces (e.g. complexity) as an additional mechanism for the identifying what constitutes relevant work dependencies among development teams and organize them accordingly. It is important to highlight that recent work has shown that logical dependencies among software modules are a major driver of work dependencies among developers [11]. Our work complements such research by also highlighting the role of the complexity associated with those logical dependencies.

4.2 Implications for Tool Design[2]

4.2.1 Improving Awareness of Interface and API Complexity

Our results also have important implications for tool design which are relevant to both practitioners and researchers. Automatic evaluation of the complexity of interfaces and APIs[3] is a particularly relevant area because since the information about interface complexities give developers the opportunity to focus on specific interfaces as well as to become aware of the implications of particular interfaces on their work. For instance, more complex interfaces could be the ones that should be more carefully documented. Another approach that we believe will benefit from considering interface complexity measures are tools that provide usage information. For instance, source code search engines like Sourcerer [2] and tools like Mica [51] and MAPO [57] that provide API examples and patterns, respectively, could be augmented with complexity information so that they could indicate to an user which examples or patterns are less complex, and, as our results indicate, less likely to introduce bugs. This has the potential to improve code search effectiveness, but further research needs to validate that.

Our results provide the empirical basis for tools to focus on performing automatic evaluation of interfaces in terms of their complexity. However, to the best of our knowledge, Metrix [17] is one of the few tools providing automatic evaluation .of Java-based interfaces and APIs regarding their complexity. Therefore, another potential research direction is on the automatic evaluation of interfaces and APIs regarding their complexity, and as discussed in the previous section, the associated metrics to do so.

Another set of development tools that could be enhanced by considering interface complexity measures are those tools related to providing context-aware information. For instance, the work on directives [15] could be extended with a color-based notion of the complexity of the interface. Then, a developer would not only acquire information about particular requirements and assumptions on the interface, but also have an indication of the associated complexity. Such a mechanism would raise awareness of the complexity of the APIs and its potential implications on the individual's work. Certainly, it remains a research question for the future the determination of whether such improvements on awareness are associated with a reduction on the detrimental impact of such complexity.

The stability of APIs is an issue of great importance in software development organizations because of the host of negative implications that changes (particularly those that go unidentified) have on quality and productivity. CatchUp [31] is a tool that supports API maintenance by recording API changes and

[2] As mentioned before, our discussion on this section addresses both interfaces and APIs, since research tools have focused more on APIs than on interfaces.

[3] In contrast to API usability inspection methods as presented by [22] and API design guidelines like [8, 20].

using this information to automatically update clients' code. Again, our results could be used to extend CatchUp by providing information about the complexity of the interfaces being changed, or better yet, our results demonstrate that it would even be important to suggest developers *what* to change in a particular API.

4.2.2 Interface and API Complexity and Usage

Another research area regarding APIs that could benefit from our results is the identification of API "hots spots" [54]. In this case, in addition to identifying the most used parts of an API, a tool could also present information regarding the complexity of these parts. This would allow identifying whether the most complex, and error-prone, parts of an API are also the most used. In fact, we extended the Metrix tool to perform this type of analysis. We used Metrix's integration to the Sourcerer code engine to support this feature. Figure 2 below presents an example of the antlr API. The size of the squares is mapped to calls from other projects that exist in the Sourcerer database. Color is mapped to complexity. Therefore, large green squares suggest a well-designed API, one that most calls are made to classes that are not complex.

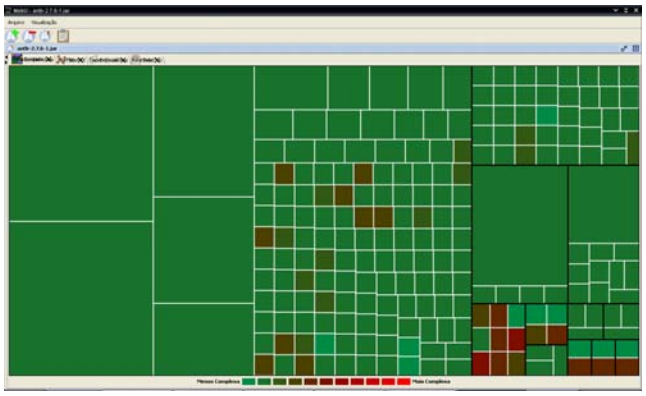

Figure 2 - Analysis of complexity and usage of the antlr API.

On the other hand, Figure 3 suggests that the svnkit API is not as well-designed: it is possible to observe that this API has classes called very often (somewhat large squares) that have some complexity (they are not so green in the complexity scale). In this case, what we are suggesting is that API design and maintenance can be improved by combining *complexity* (and, consequently "buggyness") information with *usage* information. In this case, we are talking about external API clients, i.e. projects that are different from the API under analysis. In contrast, section 3.5 reports the usage of the interfaces in the same systems. We argue that both approaches are interesting research paths to be pursued.

In general, our analyses demonstrate the viability of using interface complexity information for mechanisms that development tools could use for "tagging" or "highlighting" particular source code files or portions of code. Once parts of the source code are marked, several tools could extract this information to support API design, maintenance, and usage.

4.3 Limitations

Our work has limitations worth noticing. First, both our systems were mostly developed using the C programming language (with some minor amount of code in C++). Systems developed using programming languages with technical properties very different from C and C++ might exhibit a different impact of the measures of interface complexity on failure proneness, an issue that future research should examine. Second, the measures proposed by Bandi and colleagues [4] do not consider data elements as interfaces. As discussed in section 3.2.2, we explore two alternative methods of extending the measure and the results were similar. However, we think that redefining the interface size measure requires further evaluation. Unfortunately, we did not have access to a third system that also made heavy use of global variables as interfaces. Third, recent research has examined the impact of organizational factors on failure proneness such as organizational structure [42] and different types of work dependencies [12]. Unfortunately, we did not have access to such data for both systems.

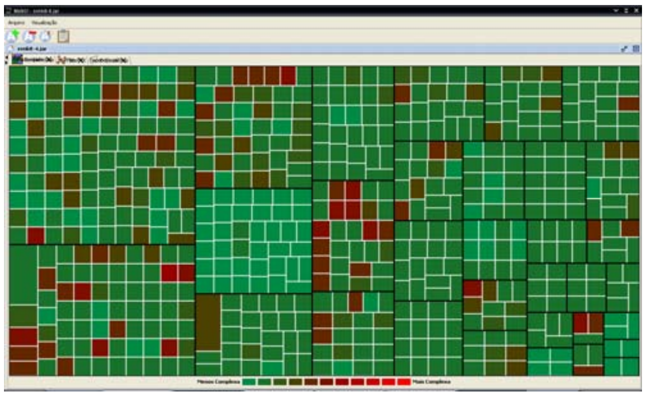

Figure 3 - Analysis of complexity and usage of the svnkit API.

5. CONCLUSIONS

In this paper, we examine the relative impact of interface complexity on the failure proneness of source code files. We performed our analysis using data from two large-scale systems from two distinct software companies. We used Bandi's et al. interface complexity metrics and compared our results with several other predictors of bug failures.

Our analyses showed that increases in the complexity of interfaces are associated with increases the failure proneness of source code files, i.e., files containing more complex interfaces are more likely to have bugs. We also extended a software tool, Metrix, to help our analysis and illustrated how such work can uncover design decisions. Building on our empirical results, we explore several paths for improving development tools aimed at assisting developers to cope with the detrimental impact of interface complexity.

6. REFERENCES

[1] Baldwin, C.Y. and Clark, K.B. *Design Rules: The Power of Modularity*. MIT Press, 2000.

[2] Bajracharya, S., Ossher, J., and Cristina Lopes 2009. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, infrastructure, Tools and Evaluation* (May 16 - 16, 2009). IEEE Computer Society, 1-4.

[3] Ball, T. and Rajamani, S.K. Automatically Validating Temporal Safety Properties of Interfaces. In *Lecture Notes in Computer Science*, Dwyer, M.B. (Ed), Springer, 2001.

[4] Bandi, R.K., et al. Predicting Maintenance Performance Using Object-Oriented Design Complexity Measures. *IEEE Trans. on Soft. Eng.*, 29, pp. 77-87, 2003.

[5] Banker, R.D., Davis, G.B., and Slaughter, S.A. Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science*, 44, 4, 1998, pp. 433-450

[6] Basili, V.R. and Perricone, B.T. Software Errors and Complexity: An Empirical Investigation. *Comm. of the ACM*, 12, pp. 42-52, 1984.

[7] Bass, L.J. et al. *Software Architectures in Practice, 2nd Edition*. Addison-Wesley Professionals, 2003.

[8] Bloch, J. How to design a good API and why it matters. OOPSLA Companion 2006: 506-507.

[9] Briand, L.C., et al. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *The Journal of Systems and Software*, 2000.

[10] Buschmann, F. et al. *Pattern Oriented Software Architecture, Volume 4*. Wiley, 2007.

[11] Cataldo, M., Herbsleb, J.D. and Carley, K.M. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proc. of the ESEM'08*, Kaiserslautern, Germany, 2008.

[12] Cataldo, M., Mockus, A., Roberts, J.A. and Herbsleb, J.D. Software Dependencies, Work Dependencies and their Impact on Failures. Forthcoming *IEEE Transactions on Software Engineering*, 2009.

[13] Conway, M.E. 1968. How do committees invent? *Datamation*, 14, 5, 28-31.

[14] Curtis, B., Kransner, H. and Iscoe, N. A field study of software design process for large systems. *Comm. of ACM,* 31, pp. 1268-1287, 1988.

[15] Dekel, U. & Herbsleb J.D. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of ICSE'09,* Vancouver, Canada, 2009

[16] de Souza, C.R.B., et al. How a Good Software Practice Thwarts Collaboration – The multiple roles of APIs in Software Development. In *Proceedings of the Conference on Foundations of Software Engineering*, pp. 221-230, 2004.

[17] de Souza, C.R.B and Bentolila, D.L.M. Automatic Evaluation of API Usability Using Complexity Metrics and Visualizations. *Proceedings of ICSE'09, New and Emerging Results*, Vancouver, Canada, 2009.

[18] de Souza, C.R.B and Redmiles, D. On the Role of APIs in the Coordination of Collaborative Software Engineering. Journal of CSCW (Forthcoming), 2009.

[19] des Rivieres, J. *How to Use the Eclipse API*. 2001 May 18, 2001; Available from: http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[20] des Rivieres, J. *Eclipse APIs: Lines in the Sand*. EclipseCon 2004; Available from: http://eclipsecon.org.

[21] Eaddy, M., et al. Do Crosscutting Concerns Cause Defects? *IEEE Trans. on Soft. Eng.*, 34, pp. 497-515, 2008.

[22] Ellis, B., J. Stylos, and B. A. Myers. The Factory Design Pattern in API Design: A Usability Evaluation. In *Proc. of* ICSE'07, Minneapolis, MN, 2007.

[23] Fenton, N.E. and Neil, M. A Critique of Software Defect Prediction Models. *IEEE Trans. on Soft. Eng.*, 25, pp. 675-689, 1999.

[24] Fenton, N.E. and Ohlsson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering,* 26, 8, 2000, pp. 797-814.

[25] Gall, H. et al. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*, pp. 190-198, 1998.

[26] Graves, T.L., Karr, A.F., Marron, J.S. and Siy, H. Predicting Fault Incidence Using Software Change History, *IEEE Trans. on Soft. Eng.*, 26, pp. 653-661, 2000.

[27] Grinter, R.E., Herbsleb, J.D. and Perry, D.E. The Geography of Coordination Dealing with Distance in R&D Work. *In Proceedings of the GROUP'99*, 1999.

[28] Halstead, M. *Elements of Software Science*. Elsevier Science Inc., 1977

[29] Hassan, A.E. and Holt, R.C. C-REX: An Evolutionary Code Extractor for C. Presented at *CSER Meeting*, Canada, 2004.

[30] Henning, M. API Design Matters. Communications of the ACM, 52 (5), 2009, pp. 46-56.

[31] Henkel, J., Diwan, A. CatchUp!: capturing and replaying refactorings to support API evolution. International Conference on Software Engineering, 2005, pp. 274-283.

[32] Herbsleb, J. D. and R. E. Grinter (1999). "Architectures, Coordination, and Distance: Conway's Law and Beyond." IEEE Software: 63-70.

[33] Herbsleb, J.D., Mockus, A. and Roberts, J.A. Collaboration in Software Engineering Projects: A Theory of Coordination. Presented at the *International Conference on Information Systems (ICIS'06),* 2006.

[34] Kawryhaw and Robilliard, Detecting Inefficient API Usage. New ideas and emergent results paper. In *Proceedings of the International Conference on Software Engineering,* pp. 183-186, 2009.

[35] Kemerer, C.F. Software Complexity and Software Maintenance: A Survey of Empirical Research. Annals for Software Engineering, 1 (1), 1995, pp. 1-20.

[36] Kiczales, G. (1996). Beyond the Black Box: Open Implementation. *IEEE Software* 13(1): 8-11.

[37] Larman, G., *Protected Variation: The Importance of Being Closed.* IEEE Software, 2001. **18**(3): p. 89-91.

[38] Li, H.F. and W.K. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering,* SE-13(6), June 1987, p. 697-708.

[39] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, SE-2 (4), 1976, p.308-320.

[40] Mockus, A. and Weiss, D. Globalization by chunking: a quantitative approach. *IEEE Software*, 18, pp. 30-37, 2001.

[41] Nagappan, N. and Ball, T. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proc. of ESEM'07*, Spain, 2007.

[42] Nagappan, N., Murphy, B., Basili, V.R. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proc. of ICSE'08*, Leipzig, Germany, 2008.

[43] Nambiar, S. and Cataldo, M. Coordination Requirements and Software Architectures: Understanding the Critical Sources of Technical Dependencies. Presented at the *2nd International Workshop on Socio-Technical Congruence,* Vancouver, Canada, 2009.

[44] Olson, G.M. and Olson, J.S. Distance Matters. *Human-Computer Interaction*, 15, 2 & 3 (2000), pp. 139-178

[45] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. of ACM*, 15, pp. 1053-1058, 1972.

[46] Perry, D.E. and Evangelist, W.M. An Empirical Study of Software Interface Faults. In *Proceedings of the International Symposium on New Directions in Computing*, Trondheim Norway, 1985.

[47] Robillard, M.P. What Makes APIs Hard to Learn? Answers from Developers. To appear in *IEEE Software - Special Issue on the Collaborative and Human Aspects of Software Engineering,* November/December 2009.

[48] Simon, H. A. The Architecture of Complexity: Hierarchical Systems. *The Sciences of the Artificial*. MIT Press, 1996..

[49] Sosa, M., Gargiulo, M. and Rowles, C.M. *Component Connectivity, Team Network Structure and the Attention to Technical Interfaces in Complex Product Development*. Working Paper, INSEAD Business School, 2007

[50] Stevens, W.P., Myers, G.J. and Constantine, L.L. Structure Design. *IBM Systems Journal*, 13, pp. 231-256, 1974.

[51] Stylos, J. and B. A. Myers (2006). Mica: A web-based search tool for finding API components and examples. In *Proceedings of VL/HCC'06* , Brighton, UK, 2006.

[52] Sullivan, K.J., et al. The Structure and Value of Modularity in Software Design. *In Proc. of FSE'01,* Vienna, Austria.

[53] Taylor, R.N et al. *Software Architecture: Foundations, Theory and Practice*. Wiley, 2009.

[54] Thummalapenta, S. and Xie, T. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *Proc. of ASE'08*, 2008.

[55] Watts, D.J. *Small Worlds: The Dynamics of Networks between Order and Randomness*, Princeton Press1994.

[56] Weyuker, E.J., "Evaluating Software Complexity Measures," *IEEE Trans. on Soft. Eng.*, 14 (9), 1988, p. 1357-1365

[57] Xie, T. and Pei, J. MAPO: Mining API Usages from Open Source Repositories. In *Proc. of MSR'06*, 2006.

[58] Zimmermannn, T. and Nagappan, N. The Predicting Defects using Network Analysis on Dependency Graphs. In *Proc. of ICSE'08, Leipzig, Germany*, 2008.

[59] Zuse, H. *Software Compleity: Measures and Methods.* Gruyter & Co, 1991.,