Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects

Ramanath Subramanyam and M.S. Krishnan

Abstract—To produce high quality object-oriented (OO) applications, a strong emphasis on design aspects, especially during the early phases of software development, is necessary. Design metrics play an important role in helping developers understand design aspects of software and, hence, improve software quality and developer productivity. In this paper, we provide empirical evidence supporting the role of OO design complexity metrics, specifically a subset of the Chidamber and Kemerer suite, in determining software defects. Our results, based on industry data from software developed in two popular programming languages used in OO development, indicate that, even after controlling for the size of the software, these metrics are significantly associated with defects. In addition, we find that the effects of these metrics on defects vary across the samples from two programming languages—C++ and Java. We believe that these results have significant implications for designing high-quality software products using the OO approach.

Index Terms—Object-oriented design, software metrics validation, object-oriented languages, C++, Java.

1 Introduction

The object-oriented (OO) approach to software development promises better management of system complexity and a likely improvement in project outcomes such as quality and project cycle time [8]. Research on metrics for OO software development is limited and empirical evidence linking the OO methodology and project outcomes is scarce. Recent work in the field has also addressed the need for research to better understand the determinants of software quality and other project outcomes such as productivity and cycle-time in OO software development [4], [21]. Of these outcomes, the importance of detection and removal of defects prior to customer delivery has received increased attention due to its potential role in influencing customer satisfaction [27] and the overall negative economic implications of shipping defective software products [32]. Hence, researchers have proposed several approaches to reduce defects in software (for e.g., see [1], [33], [39]). Suggested solutions include improvement of clarity in software design, effective use of process and product metrics, achievement of consistency and maturity in the development process, training of software development teams on tracking in-process defects, and promotion of practices such as peer reviews and causal defect analyses.

Empirical evidence in support of the effectiveness of the above approaches has also been presented [33], [34]. For example, Krishnan et al. empirically show that higher up-front investment in design helps in controlling costs as well as in improving quality [34]. It has also been shown that a number of software size-related metrics such as lines-of-code and McCabe's cyclomatic complexity are associated with defects and maintenance changes in a software system [1], [33]. Similarly, prior research has also shown that measures of testing effectiveness and test coverage can significantly explain defects [39]. However, most of these studies are primarily based on data from software developed using traditional software development methods and our understanding of the applicability of these approaches and metrics in OO development settings is limited.

Design complexity has been conjectured to play a strong role in the quality of the resulting software system in OO development environments [8]. Prior research on software metrics for OO systems suggests that structural properties of software components influence the cognitive complexity for the individuals (e.g., developers, testers) involved in their development [12]. This cognitive complexity is likely to affect other aspects of these components, such as fault-proneness and maintainability [12]. Design complexity in traditional development methods involved the modeling of information flow in the application. Hence, graph-theoretic measures [36] and information-content driven measures [30] were used for representing design complexity. In the OO environment, certain integral design concepts such as inheritance, coupling, and cohesion[1] have been argued to significantly affect complexity. Hence, OO design complexity measures proposed in literature have captured these design concepts [19], [20].

One of the first suites of OO design measures was proposed by Chidamber and Kemerer [19], [20] (henceforth, CK). The authors of this suite of metrics claim that

[1] Inheritance represents the degree of reuse of methods and attributes via the inheritance hierarchy. Coupling is a measure of interdependencies among the objects, while cohesion is the degree of conceptual consistency within an object.

• *The authors are with the University of Michigan Business School, Tappan Street, Ann Arbor, MI 48109. E-mail: {ramanath, mskrish}@umich.edu.*

Manuscript received 8 Feb. 2001; revised 12 June 2002; accepted 26 Aug. 2002.
Recommended for acceptance by C. Kemerer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 113599.

0098-5589/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

these measures can aid users in understanding design complexity, in detecting design flaws and in predicting certain project outcomes and external software qualities such as software defects, testing, and maintenance effort. Use of the CK set of metrics and other complementary measures are gradually growing in industry acceptance. This is reflected in the increasing number of industrial software tools, such as Rational Rose®, that enable automated computation of these metrics. Even though this metric suite is widely cited in literature [4], [21], [24], [25], [35], empirical validations of these metrics in real world software development settings are limited. This research studies the relationship between a subset of CK metrics and the quality of OO software measured in terms of defects, specifically those reported by customers and those identified during customer acceptance testing.

This paper presents new evidence in support of the association between a subset of CK metrics and defects. The contributions of this research are manifold. First, our study presents the effect of CK metrics on defects after controlling for software size. Some of the prior research did not account for this size effect as noted by El Emam et al. [25]. Second, we validate the association between a subset of CK metrics and defects in two current language environments, namely, C++ and Java. Authors of prior papers in this topic have raised the need for such a validation across different language settings [4] and, to our knowledge, none of the published papers have compared the results across these widely adopted languages. Third, on the methodological front, we use weighted linear regression to study the interaction effect of some of these measures on software defects. Again, to our knowledge, the interaction effect of these measures has not been studied in the past.

The organization of the rest of the paper is as follows: In the next section, we discuss prior literature on OO metrics and briefly define the CK suite of metrics. In Section 3, we present the conceptual model and the research hypotheses. Section 4 describes the research site and the data collection process and Section 5 presents the empirical model and data analyses methods. We discuss the results of the study in Section 6 and, in the final section, conclude with directions for future research.

2 Prior Literature

2.1 Development of Metrics for OO Design Complexity

Promises and challenges in OO methodology have received the attention of both researchers and practitioners. Initial research in this domain primarily focused on understanding software systems in terms of objects and their properties. For example, Wand and Weber [40] have proposed a well-supported, domain-independent modeling framework, based on Bunge's ontology, for a clear understanding of an information system [15], [16]. In this framework, they define a set of core concepts that represent a view of world as composed of objects and properties. In the mapping of this framework to software systems, objects and properties are used to describe the structure and behavior of an information system. Chidamber and Kemerer proposed the first set of OO design complexity metrics using Bunge's ontology as the theoretical basis [19]. They extended the work of Wand and Weber and defined specific measures of complexity in OO design, capturing the concepts of inheritance, coupling and cohesion.

However, the CK suite of metrics did not account for potential complexity that arises from certain other OO design factors such as encapsulation and polymorphism. Subsequent researchers proposed extensions and modification to the initial set of CK metrics highlighting these gaps. For example, Abreu proposed extensions to measure encapsulation via metrics such as the Method Hiding Factor (MHF) and the Attribute Hiding Factor (AHF), which denote the information hiding aspects of a class[2] [13], [14]. The same author also proposed a measure of polymorphism, the Polymorphism Factor (PF), which denotes the ability of OO objects to take different forms based on their usage context. Similarly, Li and Henry proposed more fine-grained extensions of the CK coupling measure via measures like Message Passing Coupling (MPC) and Data Abstraction Coupling (DAC) [35].

A clear understanding of the definitions of these complexity metrics and a promise of their relevance in improving the outcomes of software development projects led to a body of research primarily focusing on the validation of these metrics. As shown in Table 1, in this limited stream of research, CK metrics have received considerable attention. These metrics are being increasingly adopted by practitioners [21] and are also being incorporated into industrial software development tools such as Rational Rose® and Together®. The object-oriented metrics proposed by Chidamber and Kemerer [19] and later refined by the same authors [20] can be summarized as follows:[3]

1. *Weighted Methods per Class (**WMC**):* This is a weighted sum of all the methods defined in a class.[4]
2. *Coupling Between Object classes (**CBO**):* It is a count of the number of other classes to which a given class is coupled and, hence, denotes the dependency of one class on other classes in the design.[5]
3. *Depth of the Inheritance Tree (**DIT**):* It is the length of the longest path from a given class to the root class in the inheritance hierarchy.
4. *Number of Children (**NOC**):* This is a count of the number of immediate child classes that have inherited from a given class.

[2] A class is a set of objects that share a common structure and behavior [8].

[3] For detailed descriptions and intuitive viewpoints related to these metrics, please refer to [19], [20].

[4] In their initial paper, Chidamber and Kemerer suggest assigning weights to the methods based on the degree of difficulty involved in implementing them [19]. Since the choice of the weighting factor can significantly influence the value of this metric, it has remained a matter of debate among researchers. Some researchers in the past have used size-like measures, such as cyclomatic complexity of methods in their weighting scheme [24], [35]. Other researchers, including the authors of this metric, have used a weighting factor of unity in their papers on validation of OO metrics [4], [20], [21]. In this study, we also use a weighting factor of unity.

[5] As per the refinement of the original authors, inheritance-based coupling is included in the CBO metric [20]. Further, only explicit invocations (and not implicit invocations) of the constructors of other classes have been counted toward the CBO measure for a particular class. This is consistent with the viewpoint stated in the papers by the original authors [20].

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Summary of Empirical Literature on CK Metrics

Study	Dependent variable	CK Metrics tested	Size controlled for?	Summary of Results
1993 Li and Henry	Maintenance code change	All metrics in the CK suite#	Yes	Two commercial systems were studied. Five of the six metrics (except CBO) helped predict maintenance effort.
1996 Basili et al.	Fault proneness (presence/ absence of a fault in a class)	All metrics in the CK suite	No	Eight student projects were studied. WMC was correlated with defects while LCOM was not correlated with defects. CBO, DIT, NOC and RFC were correlated with defects.
1998 Binkley and Schach	Maintenance code change	Class coupling, NOC	No	Two of the four systems studied were developed using OO methods. Coupling measures were correlated with maintenance code changes due to field faults, but not NOC.
1998 Chidamber et al.	Productivity, Rework and design effort	All metrics in the CK suite	Yes	Three financial services applications were studied. High CBO and low LCOM were associated with lower productivity, greater rework, and greater design effort.
1999 Briand et al.	Fault proneness	CBO, RFC, LCOM	No	An industrial case study was performed and the three CK metrics were found to be associated with fault proneness of classes.
1999 Tang et al.	Fault proneness	WMC, RFC	No	Three real time systems were analyzed for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness.
2000 Briand et al.	Fault proneness	All metrics in the CK suite	No	Eight student projects were studied. Classes with higher WMC, CBO, DIT, and RFC were more fault-prone, while classes with more children (NOC) were less fault-prone. LCOM was not associated with defects.
2000 Cartwright and Shepperd	Defect density (Testing and Post-release defects per line of code)	DIT, NOC	No	A medium-sized telecommunication system was studied. Both DIT and NOC were found to influence defect density.
2001b El Emam et al.	Fault proneness	All metrics in the CK suite#	Yes	A large telecommunication application was studied. Size was found to confound the effect of all metrics on fault proneness.

# The authors use cyclomatic complexity of each method as a weighting factor for the WMC metric.

5. *Response for a Class (**RFC**):* This is the count of the methods that can be potentially invoked in response to a message received by an object of a particular class.
6. *Lack of Cohesion of Methods (**LCOM**):* A count of the number of method-pairs whose similarity[6] is zero minus the count of method pairs whose similarity is not zero.

2.2 Empirical Literature on CK Metrics

The body of empirical literature linking CK metrics to project outcomes is growing. A brief summary of some key research in this literature is presented in Table 1. In a study of two commercial systems, Li and Henry [35] explored the link between several OO design metrics (including metrics from the CK suite) and the extent of code change, which they used as a surrogate measure for maintenance effort. Similarly, based on an investigation of several coupling measures (including CBO) and the NOC metric of the CK suite in two university software applications, Binkley and Schach [7] found that the coupling measure was associated with maintenance changes made in classes due to field failures.

Based on a study of eight medium-sized systems developed by students, Basili et al. [4] found that several of the CK metrics were associated with fault proneness of classes. In a commercial setting, Chidamber et al. [21] observed that higher values of the coupling and the cohesion metrics in the CK suite were associated with

[6] Similarity refers to the sharing of member variables by the methods.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

reduced productivity and increased rework/design effort. Analyzing a medium-sized telecommunication system, Cartwright and Shepperd [17] studied the inheritance measures from the CK suite (DIT, NOC) and found that both these measures were associated with defect density of classes.

On similar lines, initial validation studies on CK metrics by Briand et al. [11], [12] and Tang et al. [38] indicated that several design metrics from the CK suite were positively associated with fault proneness of classes. Noting that several of these prior studies had not controlled for class sizes, El Emam et al. [25] examined a large C++ telecommunication application and provided evidence for the argument that the size of the software may confound the effect of most OO design metrics on defects. Their results indicate that, after controlling for the size of the software, the residual effects of most CK metrics (except for coupling and inheritance metrics) on defect proneness are not significant. Since this finding is contrary to our understanding from prior studies of the role of size as a mediating factor in detecting defects or other project outcomes [21], [35], it needs further validation. In addition, since very few studies have analyzed the role of these metrics in multiple languages, we need further analyses to understand potential language specific differences.

2.3 Metrics Analyzed in This Study

Though the original suite of CK design metrics has six metrics [19], [20], we use only three of these in our model, WMC, CBO, and DIT. The lack of applicability of other metrics in the CK suite to our model and the potential difficulty in computing these measures led us to exclude them.

The NOC metric represents the impact a certain class may have on child classes, which inherit methods from it. Since the focus of our research is on understanding the determinants of defects occurring in a given class and not on defects in its child classes, there was no strong rationale supporting the significant role of the NOC metric of a given class in determining defects. Hence, we did not include this metric in our model and analyses even though we collected this metric. This choice is also in line with the theory and findings of certain prior defect models in the literature [38].

The RFC metric requires knowledge of all the individual messages[7] initiated by a given class. More specifically, this metric calls for the computation of all methods potentially executed in response to a message received by an object of a given class. Given the size of the software system investigated in our study, the complete details on all the individual messages initiated by objects of all the classes were not accessible to us.[8] Hence, we could not include this metric in our model. However, since this metric has also been found to be highly correlated with the WMC and the CBO metric in earlier studies [21], both of which we have included in our study, the shortcomings arising from exclusion of the RFC metric may be limited.

Some valid arguments have been raised by past researchers regarding an ambiguity in the definition of the LCOM metric [4]. It has been identified that the original definition of this metric truncates all values below zero, and this truncation limits the metric's ability to fully capture the lack of cohesion. It is possible that the truncation of values below zero in this metric may reduce the variability of this metric and limit its usefulness in explaining productivity or defects [4]. For this reason, we omitted the LCOM metric.

3 Model and Hypotheses

The primary focus of our research is on understanding the role of some of the measures defined in the CK metric suite in explaining object-oriented software defects at a class level. Although a large number of defect models have been proposed for traditional software development methodologies, our understanding of defects in object-oriented software is limited. These models have identified size as an important variable affecting defects [2], [26], [28]. The argument for including size in these defect models relates to the ability of software developers to comprehend and control the various phases and roles in developing a large software system. Because it challenges the ability of a developer to understand software through normal cognitive processes, the size of a software system alone could enhance the difficulty in comprehending a system's functionality. We believe that these determinants of size-related complexity extend to object-oriented software and necessitate an explicit control for size in studying defects. One of the few prior studies addressing defects in object-oriented software has also discussed the potential confounding effect of size on the relationship between design metrics and defect-proneness [25]. Hence, we explicitly account for the role of size in our study and hypothesize that larger classes are associated with higher number of defects.

Hypothesis 1 (H1). *Larger classes will be associated with a higher number of defects, all else being equal.*

In traditional (non-OO) design, the separation between data and procedures often makes it difficult for developers to comprehend the functionality of the system, especially when the system is large. In OO software design, cohesion[9] is argued to be an important benefit, which is expected to alleviate this difficulty in comprehension of functionality [8]. We believe that increasing the number of methods in a class may decrease its cohesion, thus increasing the likelihood of defects occurring in the class.[10] Further, the addition of more methods to a given class may also increase

[7] Message passing is the means of communication between objects in OO programs [8].

[8] The proprietary configuration management system and the presence of certain proprietary language calls in the system code also prevented us from using an automated tool for metric collection.

[9] Cohesion represents the degree of connectivity among the attributes and methods of a single class or object [8].

[10] Some researchers consider WMC to be another indicator of size [11]. In our study, we have included both WMC and size separately since the number of methods is also an indicator of cohesion [8] while size may have no relation to cohesion. Further, it could be expected that given two classes each with 100 lines of code, there could be a greater difficulty in maintaining a class with 20 methods as opposed to maintaining a class with five methods. We also acknowledge the possibility that inclusion of certain special methods such as constructors and destructors in complexity measures could artificially bias certain complexity measures such as cohesiveness (see [18]). However, these biases are not addressed in this study as we restrict the definitions of metrics to those of the original authors of these metrics [20], [21].

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

the difficulty in managing the inheritance relationship between that class and its child classes, thus raising the likelihood of occurrence of class-level defects.

Hypothesis 2 (H2). *Classes with higher values of WMC will be associated with a higher number of defects, all else being equal.*

A strong coupling between classes in an OO design can also increase the complexity of the system by introducing multiple kinds of interdependencies among the classes. Primarily, two kinds of dependencies are introduced by the existence of strongly connected classes. The first kind of dependency arises from the simple sharing of services across the classes. Increasing class coupling could make it difficult for designers and developers to maintain and manage the multiple interfaces required for sharing of services across classes.

A second kind of dependency between classes arises from the inheritance hierarchy of classes in the design. When a child class invokes a method from its parent class, the reuse of software is certainly an advantage. However, the higher the number of inherited methods and variables, the greater the difficulty for developers in comprehending and understanding the functionality of the inheriting class. Since the CBO metric captures both kinds of dependencies by considering any invocation of a method or instance variable of another class as a coupling, we hypothesize that classes with higher CBO values will be associated with a higher number of defects.

Hypothesis 3 (H3). *Classes with higher CBO values are associated with a higher number of defects, all else being equal.*

In their original work on OO metrics, Chidamber and Kemerer have argued that the number of levels above a class in the class hierarchy may, to a great extent, determine the predictability of the class's behavior [20]. They claim that the behavior of classes deep in the hierarchy could depend on the behavior of their own methods and on the behavior of methods inherited from their immediate parent and all ancestor classes. Hence, the deeper a class is placed in the hierarchy, the greater the difficulty in predicting the behavior of the class. We believe that this uncertainty about the behavior of a class may lead to several challenges in testing all the class interfaces and maintaining the class. Hence, we hypothesize that classes with high DIT values will be associated with a higher number of defects.

Hypothesis 4 (H4). *Classes with high DIT values are associated with a higher number of defects, all else being equal.*

3.1 CBO and DIT Interaction

We believe that the effect of coupling between objects on defects may actually depend upon the level of the class in the hierarchy, i.e., the DIT of a class. As noted in the hypotheses relating coupling to defects, the second kind of dependency between classes, stemming from inheritance-related coupling, might be moderated by the level of inheritance depth. In other words, the dependency between classes could be higher for classes deep in the hierarchy, i.e., with high values of DIT. Since a class deep in the hierarchy can potentially invoke methods from many of its ancestors, a relatively high CBO value at this level may add to the difficulty in comprehension and management of its attributes and behavior. Complexity arising from the moderating effect of inheritance depth could also increase the likelihood of defects occurring in a class. Hence, we hypothesize that the effect of CBO on defects may be further augmented for classes deep in the hierarchy.

Hypothesis 5 (H5). *Classes with higher CBO in conjunction with high DIT values are associated with a higher number of defects, all else being equal.*

The dependent variable for our analysis is the *defect count* for a class. Prior researchers have proposed binary classification of defect data and have used logistic regression models to measure the impact of complexity on defect-proneness [4], [11], [24]. One possible consequence of using a binary classification scheme for defect data is that a class with one defect cannot be distinguished from a class with ten defects. As a result, the true variance of defects in the data sample may not be captured in the empirical analysis. Using defect count as a dependent variable could alleviate this effect. A second consequence of using such a classification scheme, especially in models where there are potentially correlated factors such as SIZE, WMC, and CBO, is that there could be an *underestimation* of the effect of some of these factors on defects. The reasoning is as follows: Let us suppose that CBO and SIZE are highly correlated in a sample and that both these factors are correlated with defects. If two classes, one with [1 defect, 1 CBO and 10 LOC] and another with [10 defects, 20 CBO and 200 LOC], are both classified as simply having a defect (or not), it is very likely that the effect of CBO on defects is underestimated. The variations of CBO (from one to 20) and SIZE (from 10 to 200 lines of code) would be associated with no variation in defects. In some data samples, such an underestimation could result in certain metrics becoming statistically insignificant in the defect model, while in reality they might have played a role in defects. Due to such constraints, we use the actual defect count as the dependent variable in our analyses. The functional form of our model is given below.

$$\mathbf{DEFECTS} = f(\mathbf{SIZE}, \mathbf{WMC}, \mathbf{CBO}, \mathbf{DIT}, \mathbf{CBO}^{*}\mathbf{DIT})... \quad (1)$$

4 Research Site and Data Collection

Our research site is an industry leading software development laboratory that develops diverse commercial applications. The data collected for our analyses are from a relatively large B2C e-commerce application suite developed using C++ and Java as primary implementation languages. The application suite was built using client/server architecture and was designed to work on multiple operating platforms such as AIX and Windows NT. The rationale in distributing the functionality between C++ and Java classes in the system was to maximize reuse of classes that were already available from previous development efforts in the organization. The C++ classes covered the following functionality in the application suite: order and

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

billing management, user access control management, product data persistent storage, communications and messaging control, encryption, and security control. The Java classes covered the following functionality: persistent storage, currency exchange, storefront persistent data storage, user and product data management, sales tracking and order management. Prior to our data collection, the application suite was in the field for a period of 80 days. The level of abstraction and sizes of classes were relatively uniform across both C++ and Java classes. For instance, both C++ and Java classes involved functionality such as persistent storage, management of data structures, and order management. However, they did differ in terms of certain specific functionality such as memory allocation and networking, which were predominantly split between C++ and Java classes, respectively.

We collected metrics data on 706 classes in total, including 405 C++ classes and 301 Java classes, all from the same application suite, which minimizes potential systematic project-specific biases that may arise if the classes belong to different projects or systems. Even though information on which project personnel were involved in which classes was not available to us, based on discussions with project leads and managers, we gathered that the average OO-related experience of all the developers involved in this project exceeded two years. Moreover, the levels of programming experience of developers involved in the C++ and Java classes were relatively similar. The OO programming experience of personnel involved in C++ classes varied from 23 months to eight years, while the experience levels of the programmers of Java classes varied from 21 months to seven years. Each programmer was associated with more than one class.

Due to intellectual property protection issues, the fully functional system was not available to us and we could not store the software code in persistent media. However, we had viewing access to the source code and configuration management system. We also had access to design documents through the configuration management system for the duration of the study, which allowed us to capture metrics data manually. The UML design notation was used during design. Complexity measures were computed from design documents and source code. Of the complexity metrics, WMC and DIT were computed from design documents as well as code. The CBO metric was computed entirely from code and the size (LOC) measure was gathered from the functionality provided by the source code and configuration management tools.

For our dependent variable, the defect count, which includes field defects from customers and those detected during customer acceptance testing, we collected defect information at a class level by tracing the origin of each defect from defect resolution logs that were under the control of the configuration management system. This class-level defect count was later verified for correctness with the project leads and the release manager. The definitions of variables used in our analysis are given in Table 2.

The summary statistics for the above measures for C++ classes and Java classes are shown in Table 3 and Table 4, respectively. The correlation matrix for the Size, WMC, and CBO measures are presented in Table 5.

5 Empirical Model and Diagnostics

We believe that the relationship between design metrics, size, and defects at the class level is inherently nonlinear for the following reasons. First, a linear relationship between size and defects is not common, as suggested by prior studies [3], [37]. A linear relationship would imply that a class with 1,000 lines of code is likely to have 10 times the defects found in a class with 100 lines of code, which in turn is expected to have 10 times the defects found in a class with ten lines of code. As found in studies by Shen et al. [37] and Basili and Perricone [5], such a relationship is rarely observed in software applications. Second, it is often found that defects are not uniformly distributed across the modules in the system and that a few modules may account for most defects in the system [5]. Third, on similar lines, the relationship between OO design metrics and defects is not expected to be linear. For instance, a class coupled to 10 other classes is not expected to have 10 times the defects found in another class that has a single coupling. These arguments suggest that the relationship between defects, size, and metrics are not linear. To verify whether this is the case, we performed a multivariate linear regression of defects on size and complexity measures. We found that large classes behaved differently from other classes and the error variance for large classes were greater, suggesting the presence of nonlinearity and heteroskedasticity or unequal error variances for larger classes [29].[11]

The Box-Cox transformation is a useful tool to identify the appropriate nonlinear specification for a given set of dependent and independent variables [10]. The Box-Cox transformation identifies the right specification by transforming the dependent variable. If the original dependent variable is y, the Box-Cox transformation is as follows:

$$y \Rightarrow (y^\lambda - 1)/\lambda.$$

For a given random sample of data, the maximum likelihood estimate of λ is computed and used to identify the final transformation needed for the right specification. A λ value close to zero indicates the need for a logarithmic transformation; a λ value of 1 indicates that a simple linear form of the dependent variable is appropriate and a λ value of -1 indicates that a reciprocal transformation is appropriate. Since many classes in our sample had zero defects and both logarithmic and reciprocal transformations are not possible at this value, we use *1 + defects* as the dependent variable to be transformed. We applied a Box-Cox transformation to the data and identified that the maximum likelihood estimate of λ was close to –1 for both C++ and Java samples, necessitating a reciprocal transformation. Hence, the empirical model specification estimated in our analysis is as follows:

$$\begin{aligned} &1/(1 + DEFECTS) \\ &= \beta_0 + \beta_1(Size) + \beta_2(WMC) + \beta_3(CBO) \\ &+ \beta_4(DIT) + \beta_5(CBO * DIT) + \varepsilon. \end{aligned} \tag{2}$$

[11] However, the directions (signs) of the coefficients of the linear model were consistent with the nonlinear model arrived at later.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Variable Descriptions

Variable	Description
DEFECTS	The defect count is computed at the class level and includes the aggregate numbers of field defects and customer acceptance testing defects attributed to each class.
SIZE	It is a count of non-commented non-blank source lines of code in each class[#].
WMC	This measure is an aggregate count of the number of methods in each class [20]. This count also includes constructors and destructors[##] of a class.
CBO	This variable for each class is a count of the number of other classes to which a given class is coupled. A class is said to be coupled to another class if it instantiates a member variable or invokes a member method or constructors of the second class.
DIT	This is the depth of a given class in the inheritance hierarchy. It is a count of the number of class levels that are above a given class in the class hierarchy.

[#] This definition of size is consistent with industry practice. Since the definition of "a line of code" is important for the sake of consistency across classes, we used the same scripted tool that was integrated into the source code and configuration management system, to gather the lines of code count for all classes.

[##] A constructor is a method of a class that is invoked each time an object of a class is created in memory. Typically, initialization functions are performed in the constructor. The destructor of a class is invoked each time an object of a class is deleted from memory. Usually, class-specific memory clean-up activities are performed in the destructor method. For detailed explanation of these terms, please refer to [8].

In the above equation, β_0 is the constant. β_1 captures the effect of size, while β_2, β_3, and β_4 capture the effects of WMC, CBO, and DIT, respectively. β_5 captures the interaction between CBO and DIT and ε represents the error term. Note that, in the above specification, higher values of the coefficients imply association with a fewer number of defects, as a result of the reciprocal transformation of the defect variable.

5.1 Test for Pooling C++ and Java Classes

Due to inherent differences in these two samples of C++ and Java classes, it is likely that effects of design metrics on defects may be different across these two samples. The Chow test is one of the ways to identify any structural differences in a pooled sample of data [22]. We used this test to check for any structural differences in parameters of the model presented in (2) across C++ and Java. In our

TABLE 3
Summary Statistics: C++ Classes

Variable	N	Mean	Std. Dev	Min	1st Q	2nd Q	3rd Q	Max
DEFECTS	405	0.89	1.39	0	0	0	1	7
SIZE	405	163	228.1	2	58	98	181	2953
WMC	405	7.07	13.67	0	2	3	7	219
CBO	405	2.81	2.23	0	2	3	3	25
DIT	405	2.36	1.22	0	1	3	3	5

TABLE 4
Summary Statistics: Java Classes

Variable	N	Mean	Std. Dev	Min	1st Q	2nd Q	3rd Q	Max
DEFECTS	301	0.22	0.70	0	0	0	0	5
SIZE	301	135.95	215.37	1	25	59	155	1874
WMC	301	12.15	15.84	0	3	7	16	132
CBO	301	2.94	3.45	0	1	2	4	29
DIT	301	1.02	1.00	0	0	1	1	5

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
Correlations—LOC, WMC, and CBO

	C++ classes			Java classes		
	SIZE	**WMC**	**CBO**	**SIZE**	**WMC**	**CBO**
SIZE	1	-	-	1	-	-
WMC	0.359 (0.000)	1	-	0.742 (0.000)	1	-
CBO	0.161 (0.001)	-0.065 (0.193)	1	0.665 (0.000)	0.474 (0.000)	1

Pearson correlations are shown in the table, p-values are in parenthesis.

pooled sample, the Chow test rejected the null hypothesis, that data can be pooled, at all levels of significance. This result indicates that the parameter values for C++ and Java are structurally different and supports the argument that the effect of OO design metrics on defects may vary across programming languages. Hence, we estimate the regression models shown in (3) and (4) for C++ and Java respectively. For ease of representation, we denote the regression parameters for the C++ sample as $\beta_{\mathrm{C0}}\ldots\beta_{\mathrm{C5}}$ and the regression parameters for the Java sample as $\beta_{\mathrm{J0}}\ldots\beta_{\mathrm{J5}}$, as shown below.

$$\begin{aligned}&1/(1+DEFECTS)\\&=\beta_{C0}+\beta_{C1}(Size)+\beta_{C2}(WMC)+\beta_{C3}(CBO)\\&+\beta_{C4}(DIT)+\beta_{C5}(CBO*DIT)+\varepsilon,\end{aligned}\tag{3}$$

$$\begin{aligned}&1/(1+DEFECTS)\\&=\beta_{J0}+\beta_{J1}(Size)+\beta_{J2}(WMC)+\beta_{J3}(CBO)\\&+\beta_{J4}(DIT)+\beta_{J5}(CBO*DIT)+\varepsilon.\end{aligned}\tag{4}$$

5.2 Model Specification

We estimated the empirical models given in (3) and (4) using Ordinary Least Squares regression for the C++ and Java classes. The residuals from both regression models were correlated with the size of the class, and White's test confirmed the presence of heteroskedasticity [41]. Hence, we used a Weighted Least Squares (WLS) procedure to mitigate the effect of heteroskedasticity [29]. The square root of size was used as the weighting factor in our analysis.[12] The final results of the WLS regression for C++ and Java are shown in Table 6 and Table 7, respectively.

5.3 Regression Diagnostics

We also checked for any significant effects from multicollinearity and influential observations, either of which may influence the results of our analysis. We next describe the specific tests conducted to verify the presence of these effects.

Multicollinearity. Chidamber et al. have argued that the metrics WMC and CBO may be highly correlated [21]. As noted earlier, it has also been suggested that size may be correlated with some of the CK metrics [25]. In addition, in our model, presence of the interaction term (CBO*DIT) as an explanatory factor can also lead to multicollinearity. Although it may not be possible to totally avoid multicollinearity in any data set, it is important to assess the degree to which presence of multicollinearity affects the results. We examined the data for such evidence using conditions specified in Belsley et al. [6]. The maximum condition indices for C++ and Java models were well below the critical value of 20, suggested by Belsley et al. [6]. This finding indicates the absence of any significant influence of multicollinearity in our analysis.

Influential Observations. To determine the presence of influential observations, we computed the Cook's distance for each observation [23]. The maximum Cook's distance was found to be less than two, indicating absence of influential observations.

6 Discussion of Results

6.1 Influence of Class Size

The results of the Weighted Least Squares regressions in (3) and (4) for the C++ and Java classes, depicted in Tables 6 and 7, indicate that the effect of size on defects is negative and statistically significant (β_{C1} and β_{J1} are negative and significant). Note that, in our model, this indicates that an increase in size is associated with a higher number of defects. This result supports our hypothesis H1. To a certain extent, this finding is consistent with findings of El Emam et al. [25] who suggest that size tends to confound the effect of metrics on defects. However, in contrast to [25], our sample suggests that the additional effect of metrics beyond that explained by size is statistically significant. Even though our results indicate that larger classes are associated with a higher number of defects, our results need to be interpreted with caution. This is because, if we reduce the size of all classes in the

[12] Our sample indicated that the variance of the error terms in (3) and (4) was higher for larger classes. Our weighting procedure ensures that observations with smaller variances receive larger weights in the computation of regression estimates. Of the various functional forms of size, our sample indicated that the *square root* of size was found to be the appropriate functional form for the weighting factor.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

TABLE 6
WLS Estimates for C++ Classes (Square Root of Size Used as the Weight)#

Variable	Parameter	Coefficient	Standardized Coefficient	p-value for two-sided t-tests
Intercept	β_{C0}	0.6570**		0.000
SIZE	β_{C1}	- 0.0003**	- 0.237	0.000
WMC	β_{C2}	- 0.0032*	- 0.098	0.043
CBO	β_{C3}	0.0011	0.008	0.928
DIT	β_{C4}	0.1180**	0.454	0.000
CBO*DIT	β_{C5}	- 0.0210**	- 0.431	0.001
Adjusted R^2		0.237**		

** 1% significance * 5% significance

# Please note that the negative sign of the coefficients in Table 6 and Table 7 implies positive influence on defects because of the reciprocal relationship between defects and the explanatory variables.

TABLE 7
WLS Estimates for Java Classes (Square Root of Size Used as the Weight)

Variable	Parameter	Coefficient	Standardize d Coefficient	p-values for two-sided t-tests
Intercept	β_{J0}	0.9620		0.000
SIZE	β_{J1}	- 0.0005**	- 0.676	0.000
WMC	β_{J2}	0.0015	0.138	0.076
CBO	β_{J3}	- 0.0009	- 0.018	0.891
DIT	β_{J4}	- 0.0217	- 0.094	0.245
CBO*DIT	β_{J5}	0.0079*	0.300	0.048
Adjusted R^2		0.189**		

** 1% significance * 5% significance

application, it may influence other design complexity measures. For example, if the sizes of all classes in the application are kept below threshold levels, there may be an increase in complexity metrics such as CBO. This increase in CBO may lead to a higher number of defects, as seen in the analysis of C++ classes shown in Table 6.

Our results also indicate that the effect of size varies across the two programming languages. The standardized regression coefficients for the size variable across the two models in our analysis reveal that the coefficient of size for C++ classes is 0.23, whereas for Java classes it is 0.66. This difference in the effect of size could be due to several reasons. First, complexity from a single line of code may vary across programming languages. Second, it is conceivable that the functionality of the software application across the two samples may be responsible for the difference. However, in spite of minor variations in coverage of special functions such as memory allocation and networking by the C++ and Java classes, there are significant similarities in the level of application abstraction across the C++ and Java classes, which lead us to believe that the effect of functional differences could be somewhat mitigated. Further, it could be conjectured that programmers who were involved in the development of these classes could be biasing the results. While this is certainly plausible, we find that there is a similarity in the average levels of OO experience as well as in the range of programming experience between the developers of C++ and Java classes. Therefore, we believe it is unlikely that programmer-specific factors have unduly influenced the results.

6.2 Influence of Weighted Methods per Class (WMC)

Our regression results for the C++ model, shown in Table 6, indicate that an increase in the number of methods (WMC) is associated with an increase in defects ($\beta_{C2} = -0.0032$), thus supporting our hypothesis H2. In contrast to earlier findings that the residual effect of WMC on defects after controlling for size is insignificant [25], our results indicate that, for C++ classes, the residual effect of WMC on defects after controlling for size and other complexity metrics is still significant. However, note that β_{J2} is not significant in Table 7, which indicates that our Java sample does not show support for hypothesis H2.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

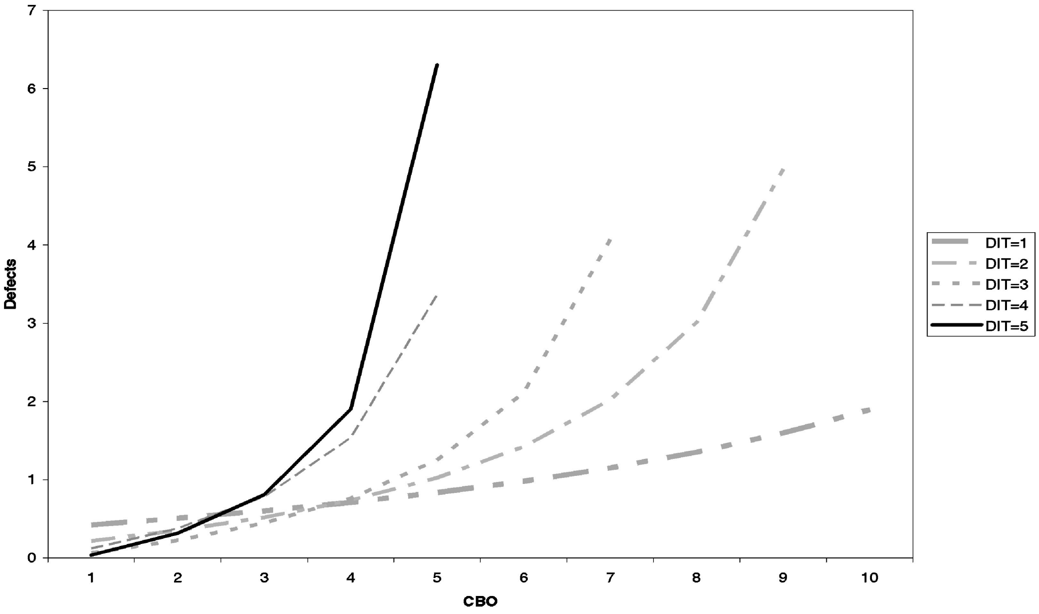

Fig. 1. Interaction Effect for C++ classes.

First, it is possible that inherent differences between these two languages may affect the influence of WMC on defects. Second, it is likely that programmer-specific and application-specific biases across the two language samples could also have played a role in our results. As noted earlier, while these biases are expected to be minimal, they cannot be entirely ruled out. Third, in our sample, the relative size of methods per class on the average is significantly lower for Java classes than for C++. The average size per method (in lines of code) for Java classes is 10.42, whereas in the case of C++ it is 28.29. This aspect, coupled with the finding that the correlation between WMC and size was relatively higher for Java classes in our sample, might also have played a role in our results (Table 5: correlation between WMC and Size was 0.65 for C++ classes and 0.74 for Java Classes).

6.3 Influence of Coupling between Objects (CBO) and Depth of inheritance (DIT)

The effects of CBO and DIT on defects in our models need to be interpreted with care because of the presence of an interaction term between these metrics. In the presence of such an interaction term, the statistical significance and values of the first order regression coefficients (β_{C3}, β_{J3}, β_{C4}, and β_{J4}) alone are not sufficient for interpretation. The significance and value of the interaction coefficients (β_{C5} and β_{J5}) should also be considered. Further, the presence of the interaction term suggests a somewhat recursive relationship, such that the effect of either of the metric cannot be studied without first fixing the level of the other metric involved in the interaction. The process of identifying the effect of two metrics on defects under the presence of interaction terms is shown in the Appendix. To interpret the effect of CBO on defects, we take the partial derivative of the estimated regression equation with respect to CBO (expression A1 in the Appendix). As depicted in (A1), the marginal effect of a change in CBO on defects depends on the value of DIT, defects, and regression coefficients (β_{C3}, β_{C5} for C++ and β_{J3}, β_{J5} for Java).

As shown in Table 6, the coefficient β_{C3} is not significant, whereas β_{C5} is negative and significant ($\beta_{C5} = -0.021$). Since defects and DIT values are always positive, the net effect of CBO on defects when DIT and other independent variables are fixed at the mean is positive for the C++ sample. A similar analysis of the Java sample indicates that the net effect of CBO on defects having fixed DIT and other independent variables at the sample mean is negative. This result indicates support for our hypothesis H3 for C++ classes, but lacks support for the Java sample.

Likewise, when CBO and the other independent variables are fixed at the sample mean, we find that the net effect of *increasing DIT* is a *decrease* in defects. This effect is true for both C++ and Java classes, thereby indicating lack of support for hypothesis H4. Our findings are in line with Briand et al. [12], who report that increased DIT was associated with a lower fault-proneness of classes. However, the presence of the interaction term indicates that marginal analysis may help us better understand how interactions between CBO and DIT influence defects.

6.3.1 *Marginal Analysis*

Holding defects, size, and the number of methods (WMC) constant at the sample means and substituting the estimated parameters for the regression (3) and (4), we plot the effect of increase in CBO on defects at different values of inheritance depth. These plots are depicted in Figs. 1 and 2. As plotted in Fig. 1, our results for the C++ sample indicate that as the depth of a class in the inheritance hierarchy increases, the positive association between CBO and defects is stronger and nonlinear. This is evident in the increasing slope of the curves in Fig. 1, with an increase in inheritance depth (DIT). For the sample of Java classes, the plot indicates that as the depth of a class in

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

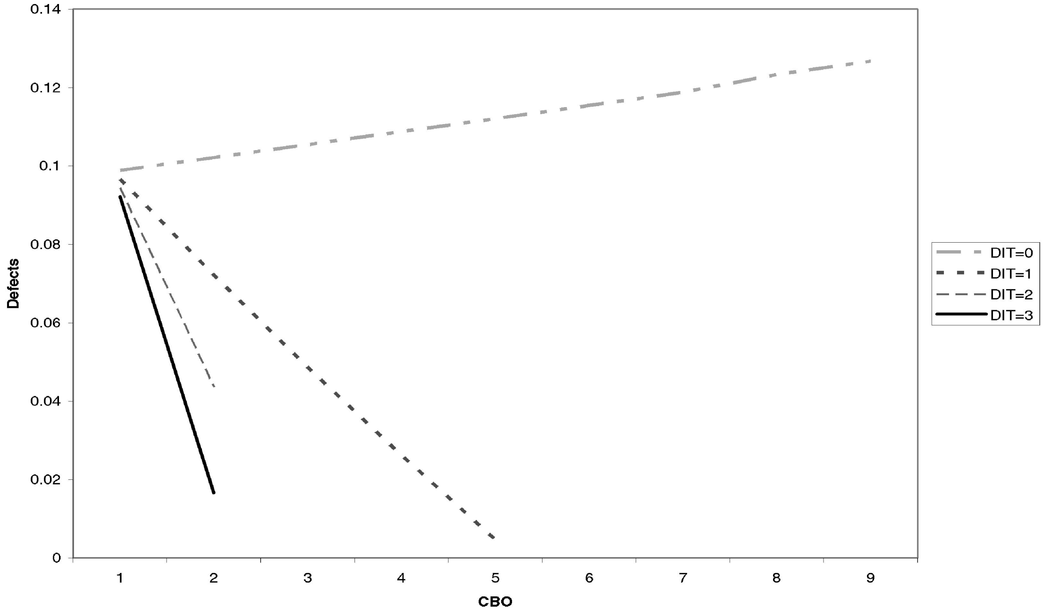

Fig. 2. Interaction Effect for Java classes.

the hierarchy increases, the effect of coupling on defects decreases. As depicted in Fig. 2, for classes at the root of the hierarchy (DIT = 0), an increase in CBO is associated with an increase in the number of defects in the class. This result is consistent with our finding in the C++ sample. However, at higher levels of inheritance depth (DIT > 0), our results show that classes with higher CBO values are associated with a fewer number of defects.[13]

6.3.2 Discussion of CBO and DIT Interactions

We have three summary results from the analyses of interactions from our dataset:

1. At the mean level of all other independent variables, C++ classes with *higher CBOs* are associated with *higher defects*, whereas they are associated with *fewer defects* for the Java sample.
2. At the mean level of all other independent variables, C++ classes as well as Java classes with *higher DITs* are associated with *higher defects*.
3. C++ classes with *high DITs as well as high CBOs* are associated with *higher defects*, whereas Java classes with *high DITs as well as high CBOs* are associated with *fewer defects*.

Several factors could play a role in these findings. They could be classified as programming language-specific factors and other general factors. We next discuss some of these factors.

Language Related Factors. Discussion of programming language-specific factors of OO development and the ability of programming languages such as C++ and Java to support them can be found in [8], [31], [9].

- *Encapsulation.* Certain programming languages may be better able to support the extent to which implementation details can be hidden. For instance, a programming language such as Smalltalk^{TM} only permits classes to have private attributes and public methods, whereas C++ allows attributes and methods to be declared public, protected or private [8] [9]. Although this choice of inheritance may provide flexibility to designers, this may also increase the design complexity and make it difficult for designers to comprehend the functionality of the system. Further research is needed to test whether such variances across languages play a role in defects.
- *Friend relationships.* The existence of *friend* classes and *friend* functions [8] in C++ could influence the role of inheritance-related coupling on defects. This is because, as a result of features such as friend functions and classes, an explicit coupling counted in the CBO metric may actually lead to a number of implicit couplings and, thus, increase the likelihood of defects due to increased complexity. Among C++ classes studied in this research, friendship was seen primarily between child classes of two functionally generic classes (of the seven in the entire application suite). The two generic classes (both with DIT of 3) concerned had 97 and 56 children, respectively. Of these, eight child classes of the first generic class and five children of the second class were involved in friend relationships owing to their need to access private attributes of the other classes. Apart from this, friendship was not used in the system. This subset of 13 classes, which had an average CBO of 5 and a DIT of 3, were found to be responsible for an average of 1.64 defects, compared to the sample average of 0.8 for C++ classes. Further research is needed to validate these results in data samples that have more classes involved in friend relationships, say, due to design constraints.

[13] The sample of Java classes used in our analysis is skewed, in that less than 25 percent of our classes exhibit higher values of DIT. There is a need to validate our findings in a richer sample of Java classes.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

- *Multiple-inheritance*. For a given class in the class hierarchy, a programming language such as C++ language does not restrict inheritance of properties to parents and ancestor classes only. This allows classes to inherit behavior and methods from more than one class, thereby increasing the complexity and likelihood of enhancing defects. Inheritance-related coupling resulting from multiple-inheritance may lead to an increase in defects. In our C++ data sample, usage of multiple-inheritance was restricted to 10 child classes of the two generic classes mentioned earlier (in the discussion of friend relationships) and another generic class (of the seven in the application suite). These 10 classes had an average of one defect compared to the sample average of 0.8. There is a need to further validate these findings under other software settings, which have classes involved in multiple-inheritance relationships.

General Factors. In addition to the above language-specific factors, several nonlanguage factors may also be the cause of the differences in results of C++ and Java samples. First, differences across the application functionality between the C++ and Java samples could have played a role in the results. On the one hand, as noted earlier, there are several functional similarities between the two data samples in our study. For instance, both samples had classes with similar functionality such as persistent storage, management of data structures, and order management. On the other hand, despite these similarities at a higher level, specific functional differences such as memory management functionality in C++ and networking functionality in Java classes could have influenced these results. Further research on complexity metrics using data samples where the same functionality is embedded in both language samples (possibly under experimental settings) may help validate our findings.

Second, personnel-specific influences could also have played a role in our results across the two samples. As discussed earlier, the profile of personnel capability and experience in the two samples used in our analysis are similar. However, experience in years is a narrow definition of capability. Future research needs to explicitly account for more relevant personnel factors while studying the role of design complexity metrics on defects. Third, our results from the Java sample were skewed. As shown in Table 2, more than 75 percent of our classes exhibit DIT values of one or zero.[14] Consequently, our results need to be interpreted with caution. Further research is needed to validate our findings on CBO and DIT for Java classes in a richer sample covering a larger range of DIT and other measures. Finally, our results may be influenced by other complexity dimensions not captured in the CK complexity metrics. These dimensions include differences in the usage of special methods across the two languages.[15] Such differences may have played a role in our results. Future studies could explicitly include complexity measures that are not captured in our analyses.

7 Conclusions

Our study enhances prior empirical literature on OO metrics by providing a new set of results validating the association between a subset of CK metrics and defects detected during acceptance testing and those reported by customers. One of our main findings is that, after controlling for size, we find that some of the measures in the CK suite of OO design complexity metrics significantly explain variance in defects.

Prior studies have also noted that, in order for OO metrics to be useful, there is an evident need for validity of these metrics across various programming languages [4], [20]. This study attempts to fill this gap by empirically validating the CK suite of OO design metrics for two popular programming languages in use today, C++ and Java. The effects of certain OO design complexity metrics, such as number of methods (WMC), coupling between objects (CBO), and inheritance depth (DIT), on defects were found to differ across the C++ and Java samples in our study.

We analyze defect counts as indicators of quality of the OO system, allowing for more variability in the dependent variable. Our approach also permits us to account for complexity effects from the interaction between two CK metrics that may play an additional role in explaining defects. To our knowledge, there is little prior research on the effects of interaction between the design metrics on defects. We validate the same CK metrics using data sets collected from software developed in two different languages suitable for object oriented software development, namely C++ and Java. Our results also indicate that the programming language might play a role in the relationship between OO design metrics and defects. The effects of such OO design metrics as number of methods (WMC), coupling between objects (CBO), and inheritance depth (DIT), on defects were found to differ across the C++ and Java samples in our study.

Like most other research in this stream, our study has several limitations. Our analyses cover only a subset of the CK suite of metrics. This research needs to be further extended to other CK measures such as LCOM and RFC. In addition, the focus of our analysis is on defects in a class. Future research may treat the effect of these complexity metrics on other measures of performance, such as the effort and time required to develop a class. Also, note that the design metrics used in our analysis address only the static aspects of OO design complexity. Further research can extend such an analysis to dynamic complexity measures such as polymorphism [13], [14]. As noted earlier, the sample of Java classes used in our analysis is skewed, in

[14] We performed a sensitivity analysis for our sample using a subset of Java classes that had DIT values of two or above. Our results indicated 1) very high levels of collinearity (condition index greater than 20, [6]) between the interaction term and DIT and 2) statistical nonsignificance of all regression coefficients, except for size, possibly due to the collinearity noted earlier.

[15] Special methods include access methods, delegation methods, constructors, and destructors [18]. For our sample, we performed additional sensitivity analyses excluding constructor and destructor invocations, and found that the magnitude of the coefficients were comparable and the direction (sign) of the coefficients of our models were consistent. This could be due to the fact that constructor invocations of other classes were accompanied by method calls in almost all cases and the resulting lack of change in the CBO count.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

that only a few classes exhibit higher values of DIT. There is a need to validate our findings in a richer sample of Java classes. Finally, the data used in our study is from a single project. Future research may want to consider validation across projects and explicitly control for other people-related factors, such as personnel capability and programmer experience in OO design, which may influence defects.

APPENDIX

To study the effect of CBO on defects, we take the partial derivative of expressions (3) and (4) and get,

$$\frac{\partial(1/1+defects)}{\partial(CBO)} = \beta_3 + \beta_5(DIT) \Rightarrow \frac{\partial(defects)}{\partial(CBO)} = -(1+defects)^2[\beta_3+\beta_5(DIT)]\ldots \tag{A1}$$

The interpretation of the coefficient for CBO is possible when the above partial derivative is computed at the mean value of defects and DIT.[16] By substituting the mean values of defects and DIT, we obtain the following coefficients for the two samples:

$$\frac{\partial(defects)}{\partial(CBO)} = 0.173$$

for the C++ sample and

$$\frac{\partial(defects)}{\partial(CBO)} = -0.011$$

for the Java sample.

This implies that for an increase in coupling by one unit, defects are expected to increase by a factor of 0.173 for the C++ sample and decrease by a factor of 0.011 for the Java sample.

Likewise, we compute the effect of DIT on defects as:

$$\frac{\partial(1/1+defects)}{\partial(DIT)} = \beta_4 + \beta_5(CBO) \Rightarrow \frac{\partial(defects)}{\partial(DIT)} = -(1+defects)^2[\beta_4+\beta_5(CBO)]\ldots \tag{A2}$$

By substituting the mean values of defects and CBO, we get:

$$\frac{\partial(defects)}{\partial(DIT)} = -0.211$$

for the C++ sample and

$$\frac{\partial(defects)}{\partial(DIT)} = -0.002$$

for the Java sample.

In other words, if we fix CBO and all other explanatory variables at the mean of the sample, increasing the depth of inheritance has a negative influence on defects. These influences vary with different levels of CBO and are also likely to change directions at certain levels.[17]

[16] Note that defects and DIT are always nonnegative.

[17] Changes of directions of coefficients indirectly suggest that there are some optimal levels at which the influences of the metrics could change from being advantageous to being detrimental.

ACKNOWLEDGMENTS

The authors would like to thank Murat Sandikcioglu, Jack Dawson, and Dragan Damnjanovic at our research site for their cooperation and support in data collection. Financial support for the study was provided from a corporate fellowship awarded to the University of Michigan and the Mary and Mike Hallman fellowship at the University of Michigan Business School.

REFERENCES

[1] F. Akiyama, "An Example of Software System Debugging," *Information Processing,* vol. 71, pp. 353-379, 1971.

[2] R.D. Banker, S.M. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Software Maintenance Costs," *Comm. ACM,* vol. 36, no. 11, pp. 81-94, 1993.

[3] R.D. Banker and C.F. Kemerer, "Scale Economies in New Software Development," *IEEE Trans. Software Eng.,* pp. 1199-1205, 1989.

[4] V. Basili, L. Briand, and W. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, pp. 751-761, 1996.

[5] V.R. Basili and B.R. Perricone, "Software Errors and Complexity," *Comm. ACM,* vol. 27, pp. 42-52, 1984.

[6] D. Belsley, E. Kuh, and R. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity.* New York: John Wiley and Sons, 1980.

[7] A. Binkley and S. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," *Proc. 20th Int'l Conf. Software Eng.,* pp. 452-455, 1998.

[8] G. Booch, *Object-Oriented Analysis and Design with Applications,* second ed. Redwood City, Calif.: Benjamin/Cummings, 1994.

[9] B. Boone, *Java(TM) Essentials for C and C++ Programmers.* Reading, Mass.: Addison-Wesley, 1996.

[10] G. Box and D. Cox, "An Analysis of Transformations," *J. Royal Statistical Soc.,* Series B, pp. 211-264, 1964.

[11] L.C. Briand, J. Wuest, J.W. Daly, and D.V. Porter, "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems," *J. Systems and Software,* vol. 51, no. 3, pp. 245-273, 2000.

[12] L.C. Briand, J. Wuest, S. Ikonomovski, and H. Lounis, "Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study," *Proc. Int'l Conf. Software Eng.,* pp. 345-354, 1999.

[13] F. Brito e Abreu, "The MOOD Metrics Set," *Proc. ECOOP'95 Workshop Metrics,* 1995.

[14] F. Brito e Abreu and W. Melo, "Evaluating the Impact of OO Design on Software Quality," *Proc. Third Int'l Software Metrics Symp.,* 1996.

[15] M. Bunge, *Treatise on Basic Philosophy: Ontology I: Furniture of the World.* Boston: Riedel, 1977.

[16] M. Bunge, *Treatise on Basic Philosophy: Ontology II: The World of Systems.* Boston: Riedel, 1979.

[17] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Software Eng.,* vol. 26, no. 7, pp. 786-796, Aug. 2000.

[18] H.S. Chae, Y.R. Kwon, and D.H. Bae, "A Cohesion Measure for Classes in Object-Oriented Classes," *Software—Practice and Experience,* vol. 30, pp. 1405-1431, 2000.

[19] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object Oriented Design," *Proc. Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA'91),* vol. 26, no. 11, pp. 197-211, 1991.

[20] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, pp. 476-493, 1994.

[21] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.,* vol. 24, pp. 629-639, 1998.

[22] G. Chow, "Tests of Equality Between Sets of Coefficients in Two Linear Regressions," *Econometrica,* vol. 28, pp. 591-605, 1960.

[23] R.D. Cook and S. Weisberg, *Residuals and Influence in Regression.* London: Chapman and Hall, 1982.

[24] K. El Emam, W. Melo, and J.C. Machado, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," *J. Systems and Software,* vol. 56, pp. 63-75, 2001.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.

[25] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.*, vol. 27, pp. 630-650, 2001.

[26] N.E. Fenton, *Software Metrics: A Rigorous Approach.* London: Chapman and Hall, 1991.

[27] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement.* New Jersey: Prentice Hall, 1992.

[28] R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program.* New Jersey: Prentice Hall, 1987.

[29] W.H. Greene, *Econometric Analysis.* New Jersey: Prentice Hall, 1997.

[30] M. Halstead, *Elements of Software Science.* New York: Elsevier North-Holland, 1977.

[31] C.S. Horstmann, *Practical Object-Oriented Development in C++ and Java.* New York: John Wiley & Sons, 1997.

[32] C. Jones, *Software Quality: Analysis and Guidelines for Success.* London: Thomson Computer Press, 1997.

[33] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman, "An Evaluation of Some Design Metrics," *Software Eng. J.*, vol. 5, no. 1, pp. 50-58, 1990.

[34] M.S. Krishnan, C.H. Kriebel, S. Kekre, and T. Mukhopadhyay, "An Empirical Analysis of Productivity and Quality in Software Products," *Management Science,* vol. 46, pp. 745-759, 2000.

[35] W. Li and S. Henry, "Object Oriented Metrics that Predict Maintainability," *J. Systems and Software,* vol. 23, pp. 111-122, 1993.

[36] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, pp. 308-320, 1976.

[37] V.Y. Shen, T. Yu, and S.M. Thebut, "Identifying Error-Prone Software-An Empirical Study," *IEEE Trans. Software Eng.,* vol. 11, pp. 317-324, 1985.

[38] M.H. Tang, M.H. Kao, and M.H. Chen, "An Empirical Study on Object Oriented Metrics," *Proc. Sixth Int'l Software Metrics Symp.,* pp. 242-249, 1999.

[39] A. Veevers and A.C. Marshall, "A Relationship between Software Coverage Metrics and Reliability," *J. Software Testing, Verification and Reliability,* vol. 4, pp. 3-8, 1994.

[40] Y. Wand and R. Weber, "An Ontological Model of an Information System," *IEEE Trans. Software Eng.,* vol. 16, pp. 1282-1292, 1990.

[41] H. White, "A Heteroscedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroscedasticity," *Econometrica,* vol. 48, pp. 817-838, 1980.

Ramanath Subramanyam received an undergraduate degree in electronics and communication engineering from Regional Engineering College, Trichy, India. He is a doctoral candidate in the Computer Information Systems Department at the University of Michigan Business School, Ann Arbor. His current research work focuses on the role of metrics and process innovations in software management. His other research interests include software engineering management, business value of information technology, customer satisfaction in software. Prior to joining the doctoral program, he worked as a software development engineer at Siemens Public Communication Networks.

M.S. Krishnan received the PhD degree in information systems from the Graduate School of Industrial Administration, Carnegie Mellon University in 1996. He is a Mary and Mike Hallman e-Business Fellow and chairman and associate professor of computer information systems at the University of Michigan Business School. He was awarded the ICIS Best Dissertation Prize for his doctoral thesis on "Cost and Quality Considerations in Software Product Management." His research interest includes corporate IT strategy, business value of IT investments, return on investments in software development process improvements, software engineering economics, metrics and measures for quality, productivity and customer satisfaction for products in software and information technology industries. In January 2000, the American Society for Quality (ASQ) selected him as one of the 21 voices of quality for the 21st century. His research articles have appeared in several journals including *Management Science, Sloan Management Review, Information Technology and People, Harvard Business Review, IEEE Transactions on Software Engineering, Decision Support Systems, Information Week, Optimize* and *Communications of the ACM.* His article "The Role of Team Factors in Software Cost and Quality" was awarded the 1999 ANBAR Electronic Citation of Excellence. He serves on the editorial board of reputed academic journals including *Management Science* and *Information Systems Research*. Dr. Krishnan has consulted with Ford, NCR, IBM, TVS group, and Ramco Systems.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.

Authorized licensed use limited to: Queens University Belfast. Downloaded on May 07,2021 at 23:37:39 UTC from IEEE Xplore. Restrictions apply.