Journal of Engineering Science and Technology
Vol. 15, No. 2 (2020) 1092 - 1108

OBJECT-ORIENTED MEASURES AS TESTABILITY INDICATORS: AN EMPIRICAL STUDY

AMOS O. BAJEH[1,*], ONILEDE-JACOBS OLUWATOSIN[2], SHUIB BASRI[3], ABIMBOLA G. AKINTOLA[4], ABDULLATEEF O. BALOGUN[5]

[1, 2, 4, 5]Department of Computer Science, Faculty of Communication and Information Sciences, University of Ilorin, Ilorin, Nigeria
[3]Computer and Information Sciences Department, Universiti Teknologi PETRONAS, 32610 Seri Iskandar, Perak, Malaysia
*Corresponding Author: bajehamos@unilorin.edu.ng

Abstract

Software measurement is one of the management strategies for developing robust and maintainable software products. The complexities inherent in software design can be controlled using software metrics. Testability is one of the sub characteristics of the maintainability attribute of software and it is the desire of quality assurance to ensure that software components are easily testable. This study empirically investigates the suitability of software metrics as indicators of software testability. The case study approach to empirical software engineering research is used in the study. Data are collected from six open-source object-oriented software products. The data consist of fifteen metric measurements of Java classes and their respective Junit test cases. Statistical analysis is conducted to show the relationship between measurements of the classes and that of their test cases. The results of the analysis showed that the complexity and coupling metrics of the Java classes are suitable indicators of the testability of the classes of object-oriented software. Also, the magnitude of the relationship is observed to be weak, which implies that the metrics do not completely measure the level of difficulty in the task of developing test cases for classes; that is, some other factors involved in test case development are not captured by software metrics.

Keywords: Empirical software engineering, Object-oriented software, Software measurement, Software quality attribute, Software testability.

1. Introduction

Testing is one of the essential activities in software process models to ensure quality. It is the major engagement of the quality assurance team of a software organization or development team [1-3]. Software testing is the process of exercising artefact, most especially code, in order to ensure that it meets user requirements as defined during software requirement engineering and to identify defects/bugs (if any) within software components. The proliferation of software in almost every human endeavour has led to its technological development and consequently to its complexity. Thus, an effective and efficient means of facilitating the testability of software components is valuable. Software testability as defined in the standard glossary of IEEE is the degree to which, a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met [4]. In a very similar way, the International Organization for Standardization (ISO) [5] defines testability as the attributes of software that bear on the effort needed to validate the software product.

Measuring software testability attribute is beneficial in software cost estimation and facilitation of software component refactoring to ensure optimal testability. This will consequently make the identification of defects easier and ensure that the software products are delivered in good quality- at least with little and bearable software bugs. Of more benefit is the measurement of the testability from the early stages of software development using high-level design artefacts such as UML diagram. This will reduce the re-work/effort needed to refactor software design than if testability is measured from implementation artefact such as software code [3, 6].

In spite of the benefits of software measurement in quality assurance, measuring software testability has been elusive. Very few studies have been reported on the use of software measures to estimate testability [6-11]. In 2005, Mouchawrab et al. [3], presented a comprehensive framework and postulated a set of hypotheses that can facilitate the measurement of OO software testability, but these hypotheses need to be empirically tested to establish the relationship between OO measures and software testability. Very few studies have been conducted to examine these relationships; more studies are needed to establish a consistent body of knowledge on the identification of suitable indicators of software testability especially the OO software, which is the most used approach to software development; investigating the empirical relationship between OO measures and testability using different software case studies other than the few used in the literature so far. This study empirically investigates the relationship between OO software measures and testability as a quality attribute. The hypothesis tested in this study is: there is a strong correlation between object-oriented class measures and their corresponding test case measures.

The remaining part of this paper is organized as follows: Section 2 presents a review of studies in the area of testability measurement of object-oriented software. Section 3 describes the research methodology used in the study. Section 4 presents and discusses the results of the study and stated the threats to validity and how they are mitigated. The paper is concluded in section 5 with some future directions.

2. Related Works

Several studies have been carried out to measure the testability attribute of software design. This section presents a review of these studies.

Binder [7] defined testability as the relative ease and expense of revealing software faults. The study implied that a testable system is the one that ensures reliability with a fixed testing budget and that testability is affected by six (6) factors, which are: characteristics of requirements and specification representation, Characteristics of implementation, built-inn-test capabilities, Test suite, Test support environment and Software development process. Conceptually and without empirical evidence, lack of cohesion among methods (LCOM), percentage of non-overloaded calls (OVR), percentage of dynamic calls (DYN) and Depth of Inheritance Tree (DIT) are identified in the study as suitable for assessing OO software testability.

Voas and Miller [8] reported a sensitivity analysis that involves evaluating the probability that software, in which, will fail on its next evaluation while testing if it contains a fault. The analysis is a pragmatic approach where mutant version (software seeded with faults) of software is repeatedly executed during testing and the likelihood that the seeded faults are detected is determined. A software component with the likelihood of fault detection is a potential site for rigorous testing.

Bache and Mullerburg [12] measured testability as the minimum number of test cases required for full testing with the assumption that full coverage testing is achievable. The study was based on the control flow coverage in software components and thus, used the control-flow based coverage testing criterion.

Baudry et al. [9] proposed a model that measure testability using the interaction of classes in OO software. Class interactions are identified in UML class diagrams as dependency paths. The existence of two or more distinct paths between two classes in a class diagram requires an increased effort for testing. Using hypothetical examples, the authors demonstrated the use of a proposed metric to measure testability in terms of the number of test cases required for testing software; no empirical evidence to support the assertion that the metric is a suitable measure for testability.

Jungmayr [10] estimated testability as the level of dependencies between classes; the more the dependencies, the more the test cases that will be required to test the increasing interfaces between the depending components. The study specified some metrics, such as the average number of components that a component depends on either directly or transitively, as a surrogate measure to identify potential components that will require more testing effort. Such components are potential candidates for refactoring.

Briand et al. [13] reported the use of instrumented contracts to improve the testability of OO software. Contracts are class invariants and operation pre and postconditions. Contracts improve testability by increasing the likelihood of detecting that there is/are faults and identifying the location of the faults in the software. The authors presented a case study in which, a large percentage of faults were identified by using instrumented contracts. Contracts reduce the effort and consequently improve testability by reducing the number of locations (in the form of methods and lines of code) that need to be examined before locating the source of faults.

Bruntink and van Deursen [11] estimated testability as the number of test cases and the effort required to develop the test cases. The study presented an empirical analysis that showed a correlation between class metrics (such as Number of

Methods in a class) and testability measured as the number of test cases and LOC per class (of test cases). Although the study was limited to class level, it reported that there is no relationship between inheritance-based metrics such as DIT and testability -this assertion cannot be conclusive since there is little or no consideration of inheritance relationship in the development of test cases for the sample software used in the empirical analysis.

Mouchawrab et al. [3] presented a hierarchical OO software testability framework that can facilitate the assessment of testability from UML diagrams. The study specified a set of hypotheses that can be empirically tested to establish the relationships between OO high-level design measures and testability attributes. The existence of such relationships will facilitate the measurement or estimation of testability early in the software development process and thus, reduce the cost of refactoring. The framework proposes the decomposition of testing into sub-activities, which are subsequently decomposed into constituent sub-activities until a leaf attribute that can be measured as a metric is reached. The study enumerated twenty (20) operational hypotheses that relate software design attributes and testability. The hypotheses are based on testing activities that are effort intensive: specifying test cases, developing drivers, developing stubs and developing oracles. Also, several OO high-level design measures that can be collected from UML diagrams and some other artefacts are defined. These measures conceptually have an impact on software testability. Furthermore, the applicability of such measures after establishing their relationship with testability was discussed. Further work is needed to empirically test the hypotheses to determine the relationship between the measures and testability.

Bruntink and van Deursen [14] identified and evaluated a set of OO software measures for the assessment of the testability of Java classes. The metrics evaluated are Depth of Inheritance Tree (DIT), Fan-Out (FOUT), Lack of Cohesion among Methods (LCOM), Line of Code per Class (LOCC), Number of children (NOC), Number of Fields (NOF), Number of Methods (NOM), Response for Class (RFC) and Weighted Methods per Class (WMC). These metrics were correlated against surrogate metrics for testability: lines of code for Class (dLOCC) and Number of Test Cases (dNOTC) measured from the test case of four (4) Java-based software. The results of the analysis varied across the four (4) sample software used for the study. Although all the metrics considered showed some level of a strong relationship with testability, FOUT, LOCC and RFC are better indicators of testability.

Mulo [15] attempted to estimate testability attribute throughout the software development life cycle owing to the fact that testing has advanced from being an activity carried out after software development to an activity that is included throughout the life cycle of the software. He considered observability and controllability as the two main factors of testability and suggested that if these two factors are applied there is a high probability of improvement in testers for good control of software.

Singh et al. [16] predicted testability by analysing the relationship between source code metrics and test metrics at package level using Eclipse (an open-source project whose functional testing is carried out at package level) as a case study. An important relationship was found between them, which showed that testability can be evaluated from source code metrics. The test metrics used are Lines of code for Test class

(TLOC), Number of Asserts (TA), Number of Test Methods (TM) and Number of test classes per test package (NT); TLOC, TM and TA are class-level metrics. The OO metrics analyzed for this study are Size, Inheritance, Coupling, Cohesion and Polymorphism measurements. The study showed that increase in size, coupling, inheritance and polymorphism increases testing efforts and thus, decreases testability.

Khatri et al. [17] presented approaches for the improvement of testability of software using software reliability growth models. The study showed that the knowledge of fault complexity and failure distribution helps improve testability and allocation of testing efforts and tools. Hence, the increase in the knowledge of bug complexities is directly proportional to the increase of the software testability.

Badri and Toure [18] investigated testability from the unit testing perspective. The study focused on showing the relationship between OO metrics and testability of classes in terms of the testing efforts required. This was achieved by carrying out an empirical investigation on the data collected from three open-source Java software that has JUnit test cases. The researchers used different metrics to evaluate the corresponding JUnit test cases in order to observe the class testability, which was consequently used to categorize testing efforts into two i.e. high & low. To determine the relationship between OO design metrics and testing efforts of classes, the logistic regression method was used. To explore the combined effect of the metrics, the multivariate logistic regression analysis was used. The result of their work showed that complexity, coupling, cohesion and size are metrics that can be used to predict unit testing effort of classes while it can be accurately tested using multivariate regression models.

Suri and Singhani [19] listed some factors that have an impact on the testability of software component among which, is Object-Oriented metrics consisting of LOC, NOC, WMC, LCOM, CBO, RFC, DIT.

Srivastava and Khaliq [20] reviewed literature that is related to software testability of OO design with the aim of gathering available knowledge on software error reduction. The result of this systematic review showed that estimating software testability from the design phase payoff since it helps to reduce the test effort, time, cost and rework. The study posits that for a software to be of high quality, it is imperative that software testability is improved to reduce the efforts required for testing object-oriented design.

Bajeh et al. [6] reported an empirical validation of OO high-level design metrics for coupling and cohesion. Coupling is measured as Number of Association (NASSOC) and Import Coupling (IC) while cohesion is measured as Cohesion Among Methods of a Class (CAMC), Normalized Harming Distance (NHD), Scaled Normalized Hamming Distance (SNHD) and Similarity-based Class Cohesion (SCC). In 2006, Bruntink and van Deursen [11] used the same testability surrogate metrics (dLOCC and dNOTC) for testability. Two sample software: Apache Ant and Jfreechart were used as case studies. The coupling metrics showed a very significant correlation with testability while only CAMC and SCC showed some level of correlation with testability. NHD and SNHD did not show the expected positive correlation with the testability surrogate metrics. This is an unexpected correlation since the highly cohesive a class is, the better the testability is expected to be. The study concluded that Number of Association (NASSOC) and CAMC are better OO high-level design metrics for estimating OO software testability at the software design phase.

Alzahrani and Melton [21] proposed a Client based Class Cohesion (CCC) metric that measures class cohesion based on client usage of its public methods. On validating CCC using three systems it was discovered that the proposed metric covered aspect not covered by existing cohesion metrics and that CCC is a good predictor of testing efforts either as an entity or combined with other cohesion metrics.

The use of prediction models to detect faulty software components has been studied [22-24]. For instance, El Emam et al. [22] developed prediction models using object-oriented metrics.

Although several studies have investigated the relationship between design metrics and software quality such as testability, there is no conclusive position on the applicability of design metrics in measuring testability of software. Thus, more studies are required to further investigate the suitability of design metrics as testability indicators, and this is a motivation for this study.

3. Methodology

This study used the case study approach to empirical research in software engineering [25, 26]. Figure 1 presents the strategy for the study. Metrics are identified from the literature and suitable sample open-source OO software systems were collected from the GitHub repository. The suitability of the software systems is determined by carrying out the test coverage analysis of the systems. This is done using the EclEmma [27] Java Code Coverage tool installed on the eclipse IDE as a plugin. In this study, the Java classes used for the analysis are those that have a minimum of 60% test coverage. This percentage is informed by the level of test coverage observed on the collected open-source software samples. Selecting software with lower test coverage will reduce the validity of the study since it will imply that software that is not rigorously tested are used. Furthermore, only one software sample will be suitable if the test coverage is increased beyond 60, and this will eliminate triangulation and thus, reduce the external validity of this study.

The hypothesis that was tested for each of the identified measures is:

H_o: There is no significant correlation between measure X and testability.

H_A: There is a [positive/negative] correlation between measure X and testability.

Testability, which is the response variable is measured as the OO measures of each of the test cases. It is expected that the Java class measurements that have a significant and strong correlation with the test cases measurements are good indicators of testability.

The statistical technique used for the empirical analysis was determined by the type of distribution of the collected data from the sample software. Although, most or all the empirical software engineering research use the non-parametric type of correlation (rho), which is not dependent on the data distribution. This study determined the actual distribution type under pining the metrics measurements by using Konglomov Smirnov (KS) test for normality.

This study used empirical data collected from open source OO software system. The open-source systems used for the empirical analysis are JFreechart, JFlex, Cobertura, JSci, JcrfSuite and Clearnlp. This software is from different domains and they all use the JUnit testing framework. Table 1 describes this software.

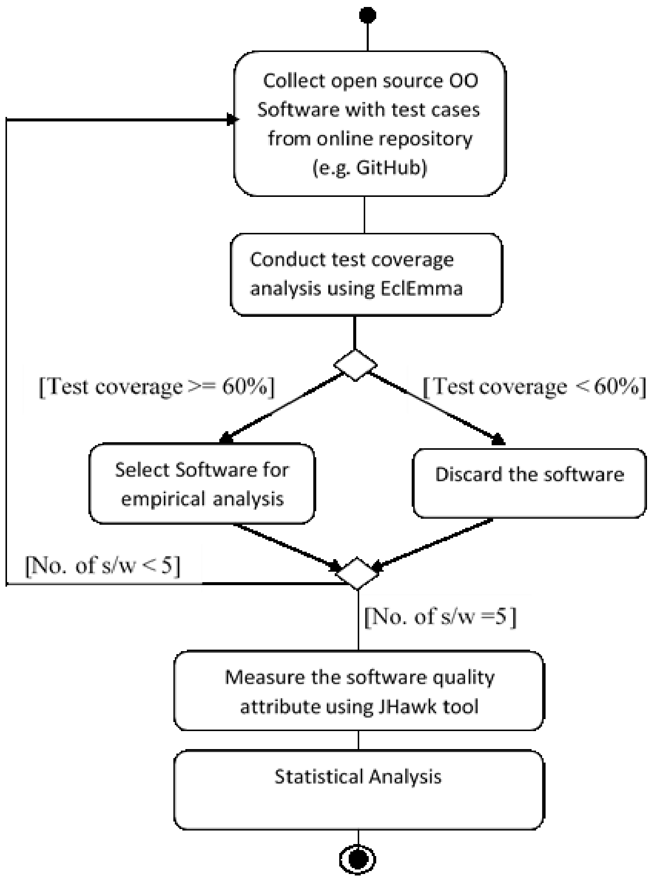

Fig. 1. Study framework activity diagram.

Table 1. Sample software.

Software	Description
JFreechart	An open-source framework used for Java programming language. Supports interactive and non-interactive charts. http://www.jfree.org/jfreechart/
JFlex	Lexical analyser generator for Java. Can also be referred to as scanner generator. https://jflex.de/
Cobertura	Java tool used for the calculation of percentages of code accessed by tests. Identifies parts of Java program that lacks test coverage. Based on Jcoverage. https://cobertura.github.io/cobertura/
JSci	Set of open source packages used to enclose scientific methods and/or principles naturally. https://Java-source.net/open-source/general-purpose/jsci-a-science-api-for-Java
JcrfSuite	Java interface for crfsuite. It gives API for loading trained model into memory and also tags sequentially in memory. https://github.com/vinhkhuc/jcrfsuite
Clearnlp	It is a software and resource for natural language processing. https://code.google.com/archive/p/clearnlp/

The empirical data are the measurement of the OO metrics [28, 29] described in Table 2. These metrics are as defined in the JHawk software measurement tool used in this study. The data are collected from the sample software by measuring the metrics from them using the JHawk software measurement tool.

These metrics were collected from the sample software using JHawk software measurement tool. JHawk is a research tool developed to measure the attributes of OO software.

Table 2. Object-oriented software metrics.

Category	OO measures	Description
Encapsulation	Number of method	Total number of methods in a Java class: This metric count the number of methods in a class
	NCO	Total number of command methods: This metric is the count of the methods that are publicly defined in a class
Coupling	RFC	Total response for class: This metric measures the total number of methods that will be invoked in a class when the class function is been performed
	CBO	Coupling between Objects: this measures the number of other classes that a class is related to
	FOUT	Fan OUT (Efferent Coupling): this measures the total number of classes that the class been measured depends upon
	F-IN	Fan-IN (Afferent Coupling): this measures the total number of classes that depend upon the class been measured
	MPC	Message Passing Coupling: This measures the number of times method invocations are made between a class and its dependents
Cohesion	LCOM	Lack of Cohesion of Methods
	LCOM2	Lack of Cohesion of Methods 2
	COH	Cohesion
	LMC	Number of Local Methods Called
Inheritance and Polymorphism	DIT	Depth of Inheritance: This measures the number of levels of classes from the root class to the class been measured
Complexity	AVCC	Average Cyclomatic Complexity: This is the average of the cyclomatic measures of the methods in a class The cyclomatic measure is the number of unique and independent execution paths in a method/code segment
	MAXCC	Maximum Cyclomatic Complexity: This is the highest cyclomatic measures of the methods in a class
	TCC	Total Cyclomatic Complexity: This is the sum of the cyclomatic measures of the methods in a class
Size	NLOC	Total Line of Code in a Class: This is the count of the number of executable statements in all the methods of a class
	HVOL	Cumulative Halstead Volume

4. Results and Discussion

The results of the test coverage analysis used to select the suitable test cases for the empirical analysis are presented in Figs. 2 and 3. The figures are samples of the results. They are screenshots of the analysis results from EclEmma tool on the eclipse platform. Figure 2 presents the test coverage of jfreeChart; it shows the Java classes with test coverages that meet up the benchmark of 60%, which make them suitable for the study. All the Java classes have coverage that ranges from 70% to 100%. For the purpose of clarity, Fig. 3 displays some test cases that are not suitable for the empirical analysis because the test coverage value is below the set benchmark of 60%.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
jfreechart-1.0.19	70.3 %	251,180	105,910	357,090
tests	95.7 %	137,841	6,208	144,049
org.jfree.chart.imagemap	100.0 %	138	0	138
DynamicDriveToolTipTagFragmentGeneratorTest.java	100.0 %	18	0	18
ImageMapUtilitiesTest.java	100.0 %	61	0	61
OverLIBToolTipTagFragmentGeneratorTest.java	100.0 %	18	0	18
StandardToolTipTagFragmentGeneratorTest.java	100.0 %	18	0	18
StandardURLTagFragmentGeneratorTest.java	100.0 %	23	0	23
org.jfree.chart.renderer	99.7 %	5,773	17	5,790
OutlierTest.java	100.0 %	129	0	129
RendererChangeDetector.java	100.0 %	17	0	17
RendererUtilitiesTest.java	100.0 %	2,211	0	2,211
AbstractRendererTest.java	99.7 %	2,661	8	2,669
LookupPaintScaleTest.java	99.0 %	413	4	417
GrayPaintScaleTest.java	99.0 %	198	2	200
DefaultPolarItemRendererTest.java	98.2 %	110	2	112
AreaRendererEndTypeTest.java	97.1 %	34	1	35

Fig. 2. A sample of the test coverage analysis of JFreechart showing test cases that are used in the analysis.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
JSci	0.1 %	5	5,638	5,643
examples/ImageProcessing	17.9 %	5	23	28
(default package)	17.9 %	5	23	28
Test.java	17.9 %	5	23	28
Test	62.5 %	5	3	8
MyFilter	0.0 %	0	20	20
JSci	0.0 %	0	57	57
(default package)	0.0 %	0	57	57
VersionApplet.java	0.0 %	0	57	57
documentation	0.0 %	0	334	334
(default package)	0.0 %	0	334	334
examples/Chaos	0.0 %	0	513	513
(default package)	0.0 %	0	513	513
CantorDustPlot.java	0.0 %	0	65	65
CatPlot.java	0.0 %	0	10	10
CatPlot	0.0 %	0	10	10
CatTransform.java	0.0 %	0	40	40
GingerbreadManPlot.java	0.0 %	0	10	10
HenonPlot.java	0.0 %	0	10	10
KochSnowflakePlot.java	0.0 %	0	158	158
MandelbrotPlot.java	0.0 %	0	210	210
StandardPlot.java	0.0 %	0	10	10

Fig. 3. Sample test coverage analysis of Cobertura showing test cases that are not used in the analysis.

Tables 3 and 4 present the descriptive statistics of the measurements of the sample software classes and their test cases respectively. They are measured using JHawk measurement tool.

Table 3. Descriptive statistics of the measurements of the classes in the OO software samples.

S/N	Metric	Min	Max	Mean	Standard deviation
1	Number of methods	0	193	16.52	18.448
2	LCOM	.0	1.0	.165	.2505
3	AVCC	.00	10.67	2.1140	1.14253
4	NOS	2	1129	117.28	148.499
5	RFC	0	199	16.78	18.813
6	CBO	0	142	12.55	15.218
7	NLOC	3	1626	173.70	211.810
8	LCOM2	0	23464	257.27	1570.213
9	MAXCC	0	55	7.99	6.237
10	NCO	0	112	6.75	10.234
11	FOUT	0	51	5.48	5.992
12	COH	.00	1.00	.2406	.19616
13	TCC	0	383	35.23	44.168
14	F-IN	0	121	7.19	12.958

Table 4. Descriptive statistics of the measurements of the test cases for the OO software samples.

Metric	Min	Max	Mean	Standard deviation
Number of methods	1	61	6.88	7.058
AVCC	1	4	1.12	.279
NOS	5	747	77.00	96.711
UWCS	1	62	7.31	7.788
RFC	1	61	6.88	7.081
CBO	1	28	4.09	3.670
NLOC	8	938	102.15	118.480
MAXCC	1	21	1.60	1.552
TCC	1	73	8.18	9.594

4.1. Correlation analysis result

The statistical analysis involves the correlation of the OO metrics of the classes against the test case measurements. Having measured the collected classes and their respective test cases using Jhawk, the measurements are tested for normality to determine, which statistical test to apply for the analysis. The Konglomorovs Smirnoff (*K-S*) test [30] for normality was used and the results showed that the measurements do not follow the normal distribution. For each metric measured the normality test returned a value of 1, which indicates that the distribution of the metrics is not a normal distribution. This implies that a non-parametric test is more appropriate for the correlation analysis. Therefore, the spearman's rho correlation test [30] was used for the correlation analysis between the class measurements and the test case measurements. Table 5 presents the correlation coefficients that are significant in the analysis done using SPSS tool. The table shows the metrics of the classes and the test cases that shows significant relationship, the correlation coefficient and the confidence level in parenthesis. The de facto p-value of < 0.05 is used to select statistically significant relationships.

Table 5. Correlation analysis.

S/N	Class metric	Test case metric	Correlation coefficient (p-value)
1	Number of methods	AVCC	-0.113 (0.035)
		MAXCC	-0.108 (0.043)
2.	LCOM	No of Methods	-0.153 (0.004)
		UWCS	-0.164 (0.002)
		RFC	-0.167 (0.002)
		CBO	-0.123 (0.021)
		NCO	-0.160 (0.003)
		FOUT	-0.124 (0.020)
		MPC	-0.115 (0.031)
		HIER	-0.109 (0.042)
		TCC	-0.132 (0.013)
3.	NOS	AVCC	-0.150 (0.005)
		LCOM2	-0.114 (0.034)
		MAXCC	-0.139 (0.009)
4.	HEFF	AVCC	-0.149 (0.005)
		MAXCC	-0.139 (0.009)
5.	RFC	AVCC	-0.111 (0.039)
		MAXCC	-0.105 (0.050)
6.	CBO	MPC	-0.136 (0.011)
7.	NLOC	AVCC	-0.156 (0.003)
		LCOM2	-0.117 (0.028)
		MAXCC	-0.144 (0.007)
8.	LCOM2	AVCC	-0.116 (0.029)
		MAXCC	-0.116 (0.030)
		COH	-0.114 (0.033)
9.	MAXCC	No of Methods	0.138 (0.010)
		AVCC	-0.146 (0.006)
		UWCS	0.117 (0.028)
		RFC	0.134 (0.012)
		NLOC	0.167 (0.002)
		MAXCC	-0.123 (0.021)
		NCO	0.153 (0.004)
		HIER	0.129 (0.016)
10.	NCO	LCOM2	-0.115 (0.031)
		COH	-0.114 (0.032)
11.	DIT	AVCC	-0.154 (0.004)
		LCOM2	0.184 (0.001)
		MAXCC	-0.136 (0.011)
		DIT	-0.115 (0.031)
12.	COH	HEFF	0.123 (0.022)
		NLOC	0.106 (0.048)
		HVOL	0.105 (0.049)
13.	HVOL	AVCC	-0.157 (0.003)
		LCOM2	-0.107 (0.045)
		MAXCC	-0.146 (0.006)
14.	TCC	AVCC	-0.138 (0.010)
		LCOM2	0.107 (0.045)
		MAXCC	-0.129 (0.016)
15.	F-IN	DIT	0.133 (0.013)
		MPC	0.134 (0.012)

4.2. Discussion

Table 5 shows that out of the 17 metrics (in Table 2) collected from the classes only 15 have a significant but weak relationship with the metric of the test cases. Thus, the null hypothesis is rejected for these metrics. The cyclomatic complexity measure, MAXCC, showed an expected positive correlation with the test case metrics: No of Methods, UWCS, RFC, NLOC, NCO and HIER. Intuitively, the positive correlation between MAXCC and the test case measurements indicates that an increase in cyclomatic complexity of the classes increases the test cases' size measures (no of methods and lines of code) and coupling (RFC). Increase in these metrics of the test cases implies an increase in the complexity of the test cases, which in turn implies that there will be an increase in the difficulty involved in the design of the test cases. Thus, to manage the testability attributes of OO classes, the cyclomatic complexity of the classes must be minimized as much as possible. Similarly, and buttressing this result, the cyclomatic complexity measure, TCC showed a significant and intuitively valid positive relationship with LCOM2, which implies that the cohesiveness of the test cases reduces (since LCOM2 measures lack cohesion) as the classes' cyclomatic complexity increases.

The weakness of the magnitude of the relationships shows that the measurement metrics do not sufficiently capture the factors that could define the level of difficulty in developing test cases and testing classes; some of the relationships cannot be explained based on metrics. Table 6 provides some explanation of the observed relationships between the classes and their test cases in terms of the sign (nature) of the relationships.

Table 6. Explanation of the relationship between class and test case measures.

S/N	Class metric	Test case metric	rho (p-value)	Explanation
1	Number of methods	AVCC MAXCC	-0.113 (0.035) -0.108 (0.043)	As the number of class methods increases the cyclomatic complexity of the test cases reduces and vice versa. A possible explanation is that if the increase in the number of methods is as a result of having more methods in a class so that the class can be cohesive and thus, cyclomatic complexity of each method reduces thus, making the average and maximum cyclomatic complexity reduce
2.	LCOM	Number of methods UWCS RFC CBO NCO FOUT MPC HIER TCC	-0.153 (0.004) -0.164 (0.002) -0.167 (0.002) -0.123 (0.021) -0.160 (0.003) -0.124 (0.020) -0.115 (0.031) -0.109 (0.042) -0.132 (0.013)	Increase in lack of cohesion of classes reduces the number of methods in the test cases and vice versa. This implies that as the cohesiveness of a class reduces (i.e., the value of LCOM increases), the class has a fewer number of methods performing several functions thus, the fewer number of methods in the test cases to test the methods. Due to the reduction in No of methods, the other metrics: UWCS, RFC, CBO, NCO, FOUT, MPC, HIER and TCC will most likely reduce thus, the negative correlations. Intuitively this relationship is not acceptable since it is desired that class cohesion is increased (i.e., the value of LCOM reduces), which will increase the values of the test case metrics such as RFC, CBO, FOUT, MPC and TCC and this is not desired because it implies an increase in the test case complexity
3.	NOS	LCOM2 AVCC MAXCC	-0.114 (0.034) -0.150 (0.005) -0.139 (0.009)	An increase in the number of statements in the classes reduces the lack of cohesion and cyclomatic complexity. This implies that as the

				number of statements increases possibly due to the increase in the number of methods to make the classes more cohesive (reducing the LCOM2 value). The splitting of methods into two or more possibly reduces the cyclomatic complexity of each method thus, the reduction in AVCC and MAXCC. This implies that increase in the number of statements in the classes does not necessarily increase the test case complexity; size (in this case NOS) is not complexity (Briand, Morasca and Basili, 1996)
4.	HEFF	AVCC	-0.149 (0.005)	Inexplicable
		MAXCC	-0.139 (0.009)	
5.	RFC	AVCC	-0.111 (0.039)	Increase in the response set of a class reduces the cyclomatic complexity of the test cases. This implies that when the number of methods called by a class (in performing its functions) increases possibly due to more methods that are cohesive, the complexity of the class methods reduces and thus, the reduction in AVCC and MAXCC
		MAXCC	-0.105 (0.050)	
6.	CBO	MPC	-0.136 (0.011)	Inexplicable
7.	NLOC	LCOM2	-0.117 (0.028)	Same as in 3 above (NOS vs. LCOM2, AVCC and MAXCC)
		AVCC	-0.156 (0.003)	
		MAXCC	-0.144 (0.007)	
8.	LCOM2	AVCC	-0.116 (0.029)	Inexplicable
		MAXCC	-0.116 (0.030)	
		COH	-0.114 (0.033)	
9.	MAXCC	Number of methods	0.138(0.010)	Increase in the cyclomatic complexity of the classes increases the size (No of methods, UWCS, NLOC, NCO) and coupling (RFC and HIER) of the test cases. This implies that managing the complexity of classes will reduce the difficulty in developing test cases. Surprisingly, the negative relationship between the MAXCC of classes and that of the test cases cannot be intuitively explained as others. This could be as a result of other factors involved in test case development, which are not captured by the metrics and thus, cannot be explained using metrics
		AVCC	-0.146 (0.006)	
		UWCS	0.117 (0.028)	
		RFC	0.134 (0.012)	
		NLOC	0.167 (0.002)	
		MAXCC	-0.123 (0.021)	
		NCO	0.153 (0.004)	
		HIER	0.129 (0.016)	
10.	NCO	LCOM2	-0.115 (0.031)	Increase in the number of methods of classes increases the cohesiveness (i.e., reduces the value of LCOM2). This is similar to the explanation in 2 above. The increase in the number of methods could be as a result of having a greater number of highly cohesive methods than a few less cohesive ones. Thus, an increase in the number of methods (NCO) increases cohesion. The relationship between NCO and COH is unexpected and inexplicable
11.	DIT	AVCC	-0.154 (0.004)	Inexplicable
		LCOM2	0.184 (0.001)	
		MAXCC	-0.136 (0.011)	
		DIT	-0.115 (0.031)	
12.	COH	HEFF	0.123 (0.022)	Increase in cohesion increases the effort (HEFF) put into development and the size of the test cases (NLOC and HVOL). The increase in cohesion could possibly cause an increase in the number of methods and consequently in the effort for developing the methods and their sizes
		NLOC	0.106 (0.048)	
		HVOL	0.105 (0.049)	
13.	HVOL	AVCC	-0.157 (0.003)	Increase in the size of the classes (HVOL) reduces the cyclomatic complexity and improves the cohesion (increase in LCOM2) of the test cases. This is possible if the increase in size is as a result
		LCOM2	-0.107 (0.045)	
		MAXCC	-0.146 (0.006)	

				of the increase in the number of methods to make the classes cohesive thus, reducing the complexity and increasing the cohesion of the test cases (i.e., reducing the value of LCOM2)
14.	TCC	LCOM2	0.107 (0.045)	Inexplicable
		AVCC	-0.138 (0.010)	
		MAXCC	-0.129 (0.016)	
15.	F-IN	MPC	0.134 (0.012)	Increase in class coupling increases the test case coupling too
		DIT	0.133 (0.013)	

Some of the results of this study agree with the findings in Bajeh et al. [6], an earlier study invalidating metrics as testability indicators. Bajeh et al. [6] observed that the relationship between test case complexity metrics and some of the cohesion metrics, specifically NHD and SNHD, is not intuitively sound. Similarly, this study observed an inexplicable relationship between LCOM2 and the test case measurements. Unlike the Bajeh et al. [6] study, this study showed no significant relationship between the class coupling metrics and the metrics of the test cases. In agreement with Bajeh et al. [6] and Bruntink and van Deursen [11], this study observed that class coupling (F-IN) has a significant relationship with test case metrics.

In conclusion, this study showed that software measurements to do sufficiently capture the factors responsible for the difficulty in developing unit test cases for classes due to the weak relationship between class measurements and that of their respective test cases. Also, in terms of the signs of the observed relationship, complexity and coupling measurements (MAXCC and F-IN) are suitable indicators of the unit testability of object-oriented classes.

4.3. Threat to validity

The observed threat to validity and how they are mitigated are as follows:

4.3.1. Internal validity

The internal validity of the study has to do with the measurement of the metrics from the open-source software used as a case study. The use of JHawk as the measurement tool prevents this threat. JHawk is a research-based tool for measuring the attributes of software components.

4.3.2. Construct validity

The relationship between the identified metrics (LCOM and MAXCC) and the testability of software components is not a course-effect relationship. They can be best represented as an association relationship since correlation does not imply causation. Thus, they are best described as surrogate metrics or indicators and not a direct measure of testability.

4.3.3. External validity

The use of open-source OO software from a different domain is to ensure the external validity of the result of this study. Although Java-based software was used, the metrics measured are not language-dependent because they can also be measured from software designed using other OO programming languages.

5. Conclusions

This study empirically investigates the suitability of object-oriented software measurements as indicators of the testability attributes of software components. The case study approach to empirical software engineering research was used. Five Java-based open-source OO software products that have Junit test cases for their classes were collected from the GitHub repository for the study. The Java classes and their corresponding test cases used have a test coverage level of at least 60%. JHawk measurement tool was used to collect the OO measures for analysis. The result of the statistical analysis showed that although the class measurements have some significant relationship with the test case measurements, the magnitude of the relationship is low. This implies that metrics do not strongly measure the level of difficulty involved in the design of test cases. Nevertheless, the cyclomatic complexity measure, MAXCC and coupling measure, F-IN showed an expected and better relationship with the test case measurements. Thus, these metrics are better indicators of the testability attribute of OO classes.

Further studies can be done to validate the results by replicating this study using more and different software samples. Other empirical software engineering research methods such as controlled experiment can be used to investigate the relationship between class metrics and test case metrics.

References

1. Pressman, R.S.; and Maxim, B.R. (2015). *Software engineering: A practitioner's approach* (8th ed.) New York, United States of America: McGraw-Hill Education.
2. Schach, S.R. (2011). *Object-oriented and classical software engineering* (8th ed.). New York, United States of America: McGraw-Hill.
3. Mouchawrab, S.; Briand, L.C.; and Labiche, Y. (2005). A measurement framework for object-oriented software testability. *Information and Software Technology,* 47(15), 979-997.
4. IEEE Computer Society. (2004). SWEBOK version 3 - download. Retrieved February 16, 2019 from https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3.
5. The International Organization for Standardization (ISO). (2014). Systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - guide to SQuaRE. ISO/IEC25000:2014(en). Retrieved March 14, 2019 from https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-2:v1:en.
6. Bajeh, A.O.; Basri, S.; and Jung, L.T. (2015). An empirical validation of coupling and cohesion metrics as testability indicators. *Lecture Notes in Electrical Engineering*, 339, 915-922.
7. Binder, R.V. (1994). Design for testability in object-oriented systems. *Communications of the ACM,* 37(9), 87-101.
8. Voas, J.M.; and Miller, K.W. (1995). Software testability: The new verification. *IEEE Software*, 12(3), 17-28.
9. Baudry, B.; Traon Le, Y.; and Sunyé, G. (2002). Testability analysis of a UML class diagram, in software metrics. *Proceedings of the Eighth IEEE Symposium.* Ottawa, Ontario, Canada, 54-63.

10. Jungmayr, S. (2002). Identifying test-critical dependencies. *Proceedings of IEEE International Conference on Software Maintenance*. Montreal, Quebec, Canada, 404-413.
11. Bruntink, M.; and van Deursan, A. (2004). Predicting class testability using object-oriented metrics. *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*. Chicago, Illinois, United States of America, 136-145.
12. Bache, R.; and Mullerburg, M. (1990). Measures of testability as a basis for quality assurance. *Software Engineering Journal,* 5(2), 86-92.
13. Briand, L.C.; Labiche, Y.; and Sun, H. (2003). Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software: Practice and Experience,* 33(7), 637-672.
14. Bruntink, M.; and van Deursan, A. (2006). An empirical study into class testability. *Journal of Systems and Software*, 79(9), 1219-1232.
15. Mulo, E. (2007). *Design for testability in software systems*. Masters Thesis. Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, the Netherlands.
16. Singh, P.K.; Sangwan, O.P.; Singh, A.P.; and Pratap, A. (2015). An assessment of software testability using fuzzy logic technique for aspect-oriented software. *International Journal of Information Technology and Computer Science,* 3, 18-26.
17. Khatri, S.; Chhillar, R.S.; and Singh, V.B. (2011). Improving the testability of object-oriented software during testing and debugging processes. *International Journal of Computer Applications,* 35(11), 24-35.
18. Badri, M.; and Toure, F. (2012). Empirical analysis of object- oriented design metrics for predicting unit testing effort of classes. *Journal of Software Engineering and Applications,* 5(7), 513-526.
19. Suri, P.R.; and Singhani, H. (2013). Object oriented software testability survey at designing and implementation phase. *International Journal of Science and Research (IJSR*), 438(4), 3047-3053.
20. Srivastava, S.; and Khaliq, M. (2014). Object oriented design for testability: A systematic review. *International Journal of Advance Research in Computer Science and Management Studies,* 2(10), 239-246.
21. Alzahrani, M.; and Melton, A. (2017). Defining and validating a client-based cohesion metric for object-oriented classes. *Proceedings of the IEEE 41st Annual Computer Software and Applications Conference (COMPSAC).* Turin, Italy, 91-96.
22. El Emam, K.; Melo, W.; and Machado, J.C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software,* 56(1), 63-75.
23. Tomaszewski, P.; Hakansson, J.; Grahn, H.; and Lundberg, L. (2007). Statistical models vs. expert estimation for fault prediction in modified code - an industrial case study. *Journal of Systems and Software,* 80(8), 1227-1238.
24. Zhou, Y.; and Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering,* 32(10), 771-789.

25. Easterbrook, S.; Singer, J.; Storey, M.-A.; and Damian, D. (2008). Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, 285-311.
26. Yin, R.K. (2009). Case study research: Design and methods sage. *Canadian Journal of Action Research,* 14(1), 69-71.
27. EclEmma 3.1.3. (2017). Java code coverage for eclipse. Retrieved February 25, 2019, from https://www.eclemma.org/.
28. Chidamber, S.R.; and Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering,* 20(6), 476-493.
29. Gupta P.; and Dhanda N. (2016). Effectiveness estimation of object oriented software: A revisit. *International Journal of Advance Research in Computer and Communication Engineering,* 4(12), 124-127.
30. Hair Jr., J.F.; Black, W.C.; Babin, B.J.; and Anderson, R.E. (2010). *Multivariate data analysis* (7^{th} ed.). Essex, England: Pearson Education Limited.