An Empirical Analysis of a Testability Model for Object-Oriented Programs

Aymen Kout, Fadel Toure and Mourad Badri
Software Engineering Research Laboratory
Department of Mathematics and Computer Science
University of Quebec at Trois-Rivières, Trois-Rivières, Quebec, Canada
{Aymen.Kout, Fadel.Toure, Mourad.Badri}@uqtr.ca

ABSTRACT
We present, in this paper, a metric based testability model for object-oriented programs. The model is, in fact, an adaptation of a model proposed in literature for assessing the testability of object-oriented design. The study presented in this paper aims at exploring empirically the capability of the model to assess testability of classes at the code level. We investigate testability from the perspective of unit testing and required testing effort. We designed an empirical study using data collected from two Java software systems for which JUnit test cases exist. To capture testability of classes in terms of required testing effort, we used different metrics to quantify the corresponding JUnit test cases. In order to evaluate the capability of the model to predict testability of classes (characteristics of corresponding test classes), we used statistical tests using correlation.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: Metrics – *product metrics*

General Terms
Measurement, Design, Experimentation

Keywords
Software Testability, Testing Effort, Model, Metrics, Relationship and Empirical Analysis.

1. INTRODUCTION
Testability is an important quality characteristic of software. IEEE [16] defines testability as the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. ISO [17] defines testability (as an important characteristic of maintainability) as attributes of software that bear on the effort needed to validate the software product.

Fenton et *al.* [10] define testability as an external attribute. Indeed, testability is not an intrinsic property of a software artifact and cannot be measured simply such as size, complexity or coupling. Testability measurement is, in fact, influenced by various factors as stated by Baudry et *al.* [1, 2]. Yeh et *al.* [33] argue also that diverse factors such as control flow, data flow, complexity and size contribute to testability. Zhao [34] states that testability is an elusive concept, and it is difficult to get a clear view on all the potential factors that can affect it.

Metrics can be used to predict testability and better manage the testing effort. Having quantitative data on the testability of a software can, in fact, help software managers, developpers and testers to [6, 14]: plan and monitor testing activities, determine the critical parts of the code on which they have to focus to ensure software quality, and in some cases use this data to review the code. A large number of object-oriented metrics (OOM) have been proposed in literature [15]. Some of these metrics (such as coupling, complexity and size) have already been used to predict testability of object-oriented systems (OOS) [6]. However, as stated by Gupta et *al.* [14], none of the OOM is alone sufficient to give an overall reflexion of software testability.

In this paper, we present a metric based testability model for object-oriented programs. The model is, in fact, an adaptation of the MTMOOD model proposed by Khan et *al.* [20] for assessing the testability of object-oriented design. The effectiveness of the MTMOOD model in predicting testability of classes at the design level has been validated using data collected from class diagrams of various medium size industrial projects. The authors showed, in fact, that testability estimated using the MTMOOD model had statistically significant correlation with the assessment determined by independent evaluators.

The study presented in this paper aims at analyzing empirically the capability of the model to predict testability of classes at the code level. We investigate testability from the perspective of unit testing and required testing effort. We designed and conducted an empirical study using data collected from two open source Java software systems for which JUnit test cases exist. To capture testability of classes in terms of required testing effort, we used different metrics to quantify the corresponding JUnit test cases. In order to evaluate the capability of the model to predict testability of classes (characteristics of corresponding test classes), we investigated empirically the relationship between the values given by the model at the code level (from the source code of software classes) and the measured characteristics of the developed JUnit test cases. We used statistical tests using correlation. The achieved results are presented and discussed in the paper.

The remainder of the article is organized as follows: Section 2 gives a survey on related work on software testability. The testability model is presented in Section 3. In Section 4, we present the used metrics to quantify JUnit test classes, describe the experimental design and discuss the statistical technique we used to investigate the relationship between the model and the characteristics of JUnit test cases. Section 5 presents the used systems. We also present and discuss in this section the obtained results. Finally, Section 6 summarizes the contributions of this work and outlines directions for further research.

2. SOFTWARE TESTABILITY
Freedman defines testability measures for software components based on two factors: observability and controllability [11]. Voas considers testability as the probability that a test case will fail if a program has a fault [30]. Voas and Miller [31] propose a testability metric based on the inputs and outputs domains of a software component, and the PIE (Propagation, Infection and Execution) technique to analyze software testability [32]. Binder [5] discusses software testability based on six factors: representation, implementation, built-in text, test suite, test support environment and software process capability.

Khoshgoftaar et *al.* address the relationship between static software product measures and testability [21], and use neural networks to predict testability from static software metrics [22]. McGregor et *al.* [26] investigate testability of OOS and introduce the visibility component measure (VC). Bertolino et *al.* [4] investigate testability and its use in dependability assessment. Le Traon et *al.* [23, 24, 25] propose testability measures for data flow designs. Petrenko et *al.* [28] and Karoui et *al.* [19] address testability in the context of communication software.

Sheppard et *al.* [29] focus on formal foundation of testability metrics. Jungmayr [18] investigates testability measurement based on static dependencies within OOS. Gao et *al.* [12] consider testability from the perspective of component-based software construction, and address component testability issues by introducing a model for component testability analysis [13]. Nguyen et *al.* [27] focus on testability analysis based on data flow designs in the context of embedded software.

Baudry et *al.* address testability measurement (and improvement) of object-oriented designs [1, 2, 3]. Metrics can, in fact, be used to locate parts of a program which contribute to a lack of testability. Bruntink et *al.* [6, 7] investigate factors of OOS testability and evaluate a set of well-known OOM with respect to their capabilities to predict testability of classes of Java software systems. They investigate testability from the perspective of unit testing. Khan et *al.* [20] propose a metric based model to assess testability of object-oriented design. Chowdhary [9] focuses on why it is so difficult to practice testability in the real world.

3. ADAPTING THE MODEL

We used in this work the Metric-Based Testability Model for Object-Oriented Design (MTMOOD) proposed by Khan et al. [20]. The model was developed to predict testability of classes at the design level by analyzing few object-oriented features in the class diagrams of software projects. The model includes three object-oriented design properties: Encapsulation, Inheritance and Coupling. Encapsulation is defined as a kind of abstraction that enforces a clean separation between the external interface of an object and its internal implementation. Inheritance is defined as a measure of the *is-a* relationship between classes. Coupling is defined as the interdependency of an object on other objects in a design.

The relative significance of each design property is weighted. The authors used a multiple linear regression analysis to get the coefficients. The effectiveness of the model in predicting design testability has been validated using data collected from class diagrams of various medium size industrial projects. The authors showed, in fact, that the overall testability estimated using the MTMOOD model had statistically significant correlation with the assessment determined by independent evaluators. The used computational formula for assessing the testability of a class at the design level is (MTMOOD): Testability = -0.08 * Encapsulation + 1.12 * Inheritance + 0.97 * Coupling.

For measuring the considered design properties, Khan et al. [20] used the following three class diagram level metrics: the metric ENM (as an encapsulation metric) which counts the number of all the methods defined in a class, the metric REM (as a reuse-inheritance metric) which counts for a class the depth of its inheritance tree in the design and the metric CPM (as a coupling metric) which counts the number of classes that a class is related to. These metrics are computed from class diagrams.

Since our work is exploratory in nature, we adapted this model to the code level by using the following source code metrics: NOO [15] which gives the number of operations in a class, DIT [8] which gives the depth of inheritance tree for a class and CBO [8] which gives the number of classes to which a given class is coupled. The following equation gives the computational formula we used for assessing the testability of a class at the source code level (MTMOOP): Testability = -0.08 * NOO + 1.12 * DIT + 0.97 * CBO.

4. EXPERIMENTAL DESIGN

The goal of this study is to explore empirically the relationship between the MTMOOP model and testability (in terms of testing effort) of classes in OOS. We estimate testability using the MTMOOP model at the class level, and limit the testing effort to the unit testing of classes. In order to achieve significant results, the data used in our experiments were collected from two open source Java software systems. This selection was essentially based on the number of classes who underwent testing using the JUnit Framework. For our experiments, we selected from each of the used systems only the classes for which JUnit test cases have been developed. In this section, we present the metrics we used to quantify the testing effort required for a class by evaluating the corresponding JUnit test class, and describe the experimental design.

4.1. Metrics related to JUnit test classes

To indicate the testing effort required for a software class (noted C_s), we used various metrics to quantify the corresponding JUnit test class (noted C_t). JUnit[1] is a simple framework for writing and running automated unit tests for Java Classes. Test cases in JUnit are written by testers in Java. A typical usage of JUnit is to test each class C_s of the program by means of a dedicated test class C_t. JUnit will report how many of the test methods in C_t succeed, and how many fail. The objective is to explore the relationship between the MTMOOP model and the characteristics of the test classes C_t.

We used the metrics TLOC and TAssert. The TLOC metric gives the number of lines of code of a test class C_t. This metric is used to indicate the size of the test suite corresponding to a class C_s. The TAssert metric gives the number of invocations of JUnit *assert* methods that occur in the code of a test class C_t. The set of JUnit *assert* methods are, in fact, used by the testers to compare the expected behaviour of the class under test to its current behaviour. This metric is used to indicate another perspective of the size of a test suite. The approach used in this paper is based on the work of Bruntik et *al.* [6]. The two metrics TLOC and TAssert have, in fact, been introduced by Bruntink et *al.* in [6, 7] to indicate the size of a test suite. Bruntink et *al.* based the definition of these metrics on the work of Binder [5]. We assume, in this paper, that these metrics are indicators of the testability of software classes. These metrics reflect different source code factors [6, 7]: factors that influence the *number of test cases required* to test the classes of a system, and factors that influence the *effort required to develop each individual test case*.

However, by analyzing the source code of the JUnit test classes of the systems we selected for our experiments, we feel that some characteristics of test classes are not captured by these two metrics (like the set of invoked methods which may indicate also the effort required for testing the methods of the class). Since our work is exploratory in nature, we decided to extend the test case metrics by using a complementary set of metrics. We used three other metrics (TNOO, TRFC and TWMPC) to capture additional characteristics of test classes. The TNOO metric (like the traditional NOO [15] metric for a software class) gives the number of operations in a test class. The TRFC metric gives the size of the response set for a test class C_t (like the traditional RFC [8] metric for a software class). The TRFC of a test class C_t is a count of its methods and the methods of other classes that are invoked by the methods of C_t. A class C_t, which provides a larger response set than another will be considered as more complex. This will reflect another perspective of the testing effort corresponding to a class under test. The TWMPC metric (like the traditional WMPC [8] metric for a software class) gives the sum of the complexities of the methods of a test class, where each method is weighted by its cyclomatic complexity. We assume that the effort necessary to write a test class C_t corresponding to a software class C_s is proportional to the characteristics measured by the used suite of test case metrics.

4.2. Data Collection

We used the MTMOOP model to estimate testability of classes for which JUnit test cases have been developed, and the test case metrics to quantify the JUnit test cases.

[1] www.junit.org

The source code and test case metrics have been computed using the Borland Together tool.

4.3. Hypotheses and Statistical Analysis

We present, in this section, the methodology of the empirical study we conducted in order to assess (explore) the relationship between the MTMOOP model and testability of classes (measured characteristics of corresponding test classes). We performed statistical tests using correlation. The null and alternative hypotheses that our experiments have tested were:

- H_0: There is no significant correlation between the MTMOOP model and testability.
- H_1: There is a significant correlation between the MTMOOP model and testability.

In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the MTMOOP model and test case metrics (chosen significance level $\alpha=0.05$). For the analysis of the collected data, we preferred a non-parametric measure of correlation. We used the Spearman's correlation coefficient. This technique, based on ranks of the observations, is widely used for measuring the degree of linear relationship between two variables (two sets of ranked data). It measures how tightly the ranked data clusters around a straight line. Spearman's correlation coefficient will take a value between -1 and +1. A positive correlation is one in which the ranks of both variables increase together. A negative correlation is one in which the ranks of one variable increase as the ranks of the other variable decrease. A correlation of +1 or -1 will arise if the relationship between the ranks is exactly linear. A correlation close to zero means that there is no linear relationship between the ranks. We used the XLSTAT software to perform the statistical analysis.

5. THE CASE STUDIES

5.1. Selected Systems

The selected systems are : ANT (www.apache.org) : a Java-based build tool, with functionalities similar to the unix "make" utility, and JFREECHART (http://www.jfree.org/jfreechart) : a free chart library for Java platform.

	SOFTWARE CLASSES			TESTED CLASSES		
SYSTEMS	#Classes	MLOC	WMPC	#TClasses	MLOC	MWMPC
ANT	713	89.85	17.1	115	153.52	30.37
JFC	496	137.73	28.09	230	231	46.08

TABLE 1: SOME CHARACTERISTICS OF THE USED SYSTEMS.

Table 1 summarizes some characteristics of the analyzed systems : number of software classes of each system, mean value of lines of code of software classes, mean value of cyclomatic complexity of software classes, number of JUnit test classes which have been developed for each system, mean value of lines of code and mean value of cyclomatic complexity of the software classes for which JUnit test classes have been developed.

The first observations that we can already make are: (1) only a subset of software classes have been tested using the JUnit Framework, (2) the pourcentage of tested classes varies from one system to another, and (3) the software classes for which JUnit test cases have been developped are in general large and complex classes.

	TLOC	TAssert	TNOO	TRFC	TWMPC
ANT	**0.325**	0.030	**0.403**	**0.507**	**0.357**
JFC	**0.283**	**0.236**	**0.260**	**0.335**	**0.265**

TABLE 2: CORRELATION VALUES BETWEEN MTMOOP AND TEST CASE METRICS.

		TLOC	TAssert	TNOO	TRFC	TWMPC
ANT	TLOC	1	**0.679**	**0.543**	**0.475**	**0.628**
	TAssert		1	0.151	0.064	**0.201**
	TNOO			1	**0.811**	**0.927**
	TRFC				1	**0.781**
	TWMPC					1
JFC	TLOC	1	**0.848**	**0.739**	**0.839**	**0.740**
	TAssert		1	**0.586**	**0.713**	**0.581**
	TNOO			1	**0.762**	**0.996**
	TRFC				1	**0.762**
	TWMPC					1

TABLE 3: CORRELATION VALUES BETWEEN TEST CASE METRICS.

5.2. Results

Table 2 and Table 3 summarize the results of the correlation analysis. Table 2 shows, for each of the subject systems and between each distinct pair of metrics (MTMOOP, test case metric) the obtained values for the Spearman's correlation coefficient r_s. We also calculated the Spearman's correlation coefficient r_s for each pair of test case metrics (Table 3). The obtained Spearman's correlation coefficients that are significant (at $\alpha=0.05$) are set in boldface in the two tables. This means that for the corresponding pairs of metrics there exist a correlation at the 95 % confidence level.

The first global observation that we can make is that the achieved results support the idea that there is a statistically significant relationship between the MTMOOP model and the used test case metrics. The obtained Spearman's correlation coefficients between the MTMOOP model and the test case metrics are overall significant (at $\alpha=0.05$) for the two selected systems (except for the system ANT between the MTMOOP model and the metric TAssert). We can reasonably reject the hypothesis H_0 and accept the hypothesis H_1. Moreover, the measures have positive correlation. As mentioned previously, a positive correlation indicates that the ranks of one variable (MTMOOP in our case) increase as the ranks of the other variable (test case metric) increase. These results are plausible and not surprising knowing the definition of the MTMOOP model and test case metrics. The other global observation that we can make is that the test case metrics are also correlated between themselves (Table 3).

For system ANT, the MTMOOP model is significantly better correlated with the test case metrics TRFC and TNOO than TLOC, TAssert and TWMPC. The correlation between the MTMOOP model and the test case metric TAssert, in particular, is not significant. For system JFREECHART, the MTMOOP model is better correlated with the test case metrics TRFC and TLOC than TAssert, TNOO and TWMPC. The results for system JFREECHART also show that the correlation values are, overall, lower than the correlation values obtained in the case of system ANT. Moreover, for both ANT and JFREECHART, the correlation values are not high.

The study performed in this paper should be replicated using many other systems in order to draw more general conclusions about the relationship between the MTMOOP model and test case metrics. In fact, there are a number of limitations that may affect the results of the study or limit their generalization:

- The obtained results are based on the data set we collected from the analyzed systems. As mentioned previously, we only used a subset of classes and corresponding JUnit test cases (see Table 1) from each of the subject systems. The study should be replicated on a large number of object-oriented systems to increase the generality of the results.
- Moreover, the classes for which JUnit test cases have been developed are relatively large and complex classes (see Table 1). This is true for the two subject systems. This may affect the results of our study in the sense that depending on the methodology followed by the developers while developing test classes and the criteria they used while selecting the software classes for which they developed test classes (randomly or depending on their size or complexity for example, or on other criteria) the results may be different. It would be interesting to replicate this study using systems for which JUnit test cases have been developed for a maximum number of classes. This will also allow observing correlation values between the MTMOOP model and test case metrics for different types of classes (small, medium and large classes).
- Also, by analyzing the source code of the developed JUnit test classes, we observed that, in many cases, they do not cover all the methods of the corresponding software classes. This may also affect the results of the study and explain why obtained correlation values are rather low in some cases.
- It is also possible that facts such as the development style used by the developers for writing test cases might affect the results or produce different results for specific applications. Indeed, the development style is different from one system to another.
- Moreover, the model as defined includes only three object-oriented properties (encapsulation, inheritance and coupling). These three properties have certainly an influence on the testability of classes. However, we believe that they do not cover all features of software classes that may affect their testability. This may explain the low values (in some cases) of the correlations. It would be interesting to explore other variants of the model including other characteristics such as size (in terms of lines of code) and complexity.

6. CONCLUSIONS AND FUTURE WORK

We presented, in this paper, a metric based testability model for object-oriented programs. The paper investigated empirically the relationship between the model and testability of classes at the code level. Testability has been investigated from the perspective of unit testing. We designed an empirical study using data collected from two open source Java software systems for which JUnit test cases exist. To capture testability of classes, we used various metrics to quantify different characteristics of the corresponding JUnit test cases. In order to evaluate the capability of the model to predict testability of classes, we used statistical tests using correlation. The achieved results support the idea that there is a statistically significant relationship between the model and the used test case metrics.

The performed study should, however, be replicated using many other systems in order to draw more general conclusions. The findings in this paper should be viewed as exploratory and indicative rather than conclusive. Moreover, knowing that software testability is affected by many different factors, it would be interesting to extend the used suite of test case metrics to better reflect the testing effort.

As future work, we plan: to replicate the study on other projects to be able to give generalized results, to extend the used test case metrics to better reflect the testing effort, and finally to explore other variants of the model.

7. ACKNOWLEDGEMENTS

This project was financially supported by NSERC (National Sciences and Engineering Research Council of Canada).

8. REFERENCES

[1] Baudry, B., Le Traon, B., Sunyé, G., Testability analysis of a UML class diagram, Proceeding of the 9th International Software Metrics Symposium *(METRICS'03)*, IEEE Computer Society, 2003.

[2] B. Baudry, B., Le Traon, Y., Sunyé, G., Jézéquel, J.M., Measuring and Improving Design Patterns Testability, Proceedings of the 9th International Software Metrics Symposium (METRICS), IEEE Computer Society, 2003.

[3] Baudry, B., Le Traon, Y., Sunyé, G., Improving the Testability of UML Class Diagrams, Proceedings of *IWoTA* (International Workshop on Testability Analysis), Rennes, 2004.

[4] Bertolino, A., Strigini, L., On the Use of Testability Measures for Dependability Assessment, IEEE Transactions on Software Engineering, VOL. 22, NO. 2, February 1996.

[5] Binder, R.V., Design for Testability in Object-Oriented Systems, Communications of the ACM, Vol. 37, 1994.

[6] Bruntink, M., Deursen, A.V., Predicting Class Testability using Object-Oriented Metrics, Fourth International Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2004.

[7] Bruntink, M., Deursen, A.V., An empirical study into class testability, Journal of Systems and Software, 2006.

[8] Chidamber, S.R., Kemerer, C.F., 1994. A Metrics suite for object Oriented Design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493, June 1994.

[9] Chowdhary, V., Practicing Testability in the Real World, International Conference on Software Testing, Verification and Validation, IEEE Computer Society Press, 2009.

[10] Fenton, N., Pfleeger, S.L., Software Metrics: A Rigorous and Practical Approach, PWS Pub. Company, 1997.

[11] Freedman, R.S., Testability of Software Components, IEEE Transactions on Software Engineering, Vol. 17(6), 1991.

[12] Gao, J., Tsao, J., Wu, Y., Testing and Quality Assurance for Component-Based Software, Artech House Publishers, 2003.

[13] Gao, J., Shih, M.C., A Component Testability Model for Verification and Measurement, Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), IEEE CSP, 2005.

[14] Gupta, V., Aggarwal, K.K., Singh, Y., A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability, Journal of Computer Science, Science Publications, 2005.

[15] Henderson-Sellers, B., Object-Oriented Metrics Measures of Complexity, Prentice-Hall, 1996.

[16] IEEE, 1990. IEEE Standard Glossary of Software Engineering Terminology, IEEE CSP, NY, 1990.

[17] ISO/IEC 9126: Software Engineering Product Quality, ISO Press, 1991.

[18] Jungmayr, S., Testability Measurement and Software Dependencies, Proceedings of the 12th International. Workshop on *Software Measurement*, October 2002.

[19] Karoui, K., Dssouli, R., Specification transformations and design for testability, Proceedings of the IEEE Global telecommunications Conference (GLOBECOM'96), London, 1996.

[20] Khan, R.A., Mustafa, K., Metric Based Testability Model for Object-Oriented Design (MTMOOD), ACM SIGSOFT Software Engineering Notes, volume 34, number 2, March 2009.

[21] Khoshgoftaar, T.M., Szabo, R.M., Detecting Program Modules with Low Testability, 11th ICSM, 1995.

[22] Khoshgoftaar, T.M., Allen, E.B., Xu, Z., Predicting Testability of Program Modules Using a Neural Network, 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 2000.

[23] Le Traon, Y. and Robach, C., Testability analysis of co-designed systems, Proceedings of the 4th Asian Test Symposium, ATS. IEEE Computer Society, Washington, DC, November 1995.

[24] Le Traon, Y., Robach, C., Testability Measurements for Data Flow Design, Proceedings of the Fourth International Software Metrics Symposium, New Mexico, November 1997.

[25] Le Traon, Y., Ouabdessalam, F., Robach, C., Analyzing testability on data flow designs, Proceedings of ISSRE'00, San Jose, CA, USA, October 2000.

[26] McGregor, J., Srinivas, S., A measure of testing effort, Proceeding of the Conference on Object-Oriented Technologies, pages 129-142. USENIX Association, June1996.

[27] Nguyen, T.B., Delaunay, M., Robach, C., Testability Analysis Applied to Embedded Data-Flow Software, Proceedings of the 3rd International Conference on Quality Software (QSIC'03), 2003.

[28] Petrenko, A., Dssouli, R., and Koenig, H., On Evaluation of Testability of Protocol Structures, In Proceedings of the Intenarional Workshop on Protocol Est Systems (IFIP), Pau, France, 1993.

[29] Sheppard, J.W., Kaufman, M., Formal Specification of Testability Metrics in IEEE P1522, *IEEE AUTOTESTCON*, Pennsylvania, August 2001.

[30] Voas, J.M., PIE: A dynamic failure-based technique, IEEE Transactions on Software Engineering, 18(8), August 1992.

[31] Voas, J., Miller, K.W., Semantic metrics for software testability, Journal of Systems and Software, Vol. 20, 1993.

[32] Voas, J.M., Miller, K.W., Software Testability: The New Verification, IEEE Software, 12(3), 1995.

[33] Yeh, P.L., Lin, J.C., Software Testability Measurement Derived From Data Flow Analysis, Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering, Italy, March 1998.

[34] Zhao, L., 2006. A New Approach for Software Testability Analysis, 28th ICSE, May 2006.