Empirical Analysis for Investigating the Effect of Control Flow Dependencies on Testability of Classes

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

***Abstract* — We present, in this paper, a new metric capturing in an integrated way different attributes of object-oriented systems. The metric uses basically control flow paths and probabilities. It captures the interactions between classes and related control. The study presented in this paper aims at exploring empirically the relationship between the proposed metric and testability of classes. We investigate testability from the perspective of unit testing. 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, we used different metrics to quantify the corresponding JUnit test cases. In order to evaluate the capability of the new metric to predict testability of classes, we used statistical tests using correlation. The achieved results provide evidence that there exist a significant relationship between the proposed metric and testability of classes.**

Keywords: Software Testability, Testing Effort, Metrics, Control Flow, Control Dependencies, Probabilities and Empirical Analysis.

I. INTRODUCTION

Software testing has an important effect on the quality of the final product. Software testing is probably the most complex task in the software development cycle. It's also a time and resources consuming process. The overall effort spent on testing depends, in fact, on many different factors including: human factors, process issues, testing techniques, tools used, and characteristics of the software development artifacts [3, 4, 9, 35, 36]. Testability is an important quality characteristic of software. Software testability is related to testing effort reduction and software quality [14]. It impacts test costs and provides a means of making design decisions [31]. Zhao [36] argues that testability expresses the affect of software structural and semantic on the effectiveness of testing following certain criterion, which decides the quality of released software. Several software development and testing experts pointed out, in fact, the importance of testability and design for testability. Moreover, Baudry et *al.* [3] argue that testability becomes crucial in the case of object-oriented systems (OOS) where control flows are generally not hierarchical, but diffuse and distributed over whole architecture.

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 [8, 15]: 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 [17]. Some of these metrics (such as coupling, complexity and size) have already been used to measure testability of OOS [8]. However, as stated by Gupta et *al.* [15], none of the OOM is alone sufficient to give an overall reflexion of software testability. Software testability is, in fact, affected by many different factors.

We presented in a previous work [2] a new metric, called *Quality Assurance Indicator* (Qi), capturing in an integrated way different attributes of OOS such as complexity and coupling (interactions between classes). The metric uses control flow paths (capturing the distribution of the control flow in a system) and probabilities. The Qi of a class C_i includes different intrinsic characteristics of the class itself, as well as the Qi of collaborating classes. The metric has, however, no ambition to capture the overall quality of OOS. Moreover, the objective is not to evaluate a design by giving absolute values, but more relative values that may be used, for example, for identifying the critical classes that will require a high testing effort to ensure software quality. The metric has been implemented for Java programs. We compared, in [2], the Qi metric using the Principal Components Analysis (PCA) method to some well-known OOM. The evaluated metrics were grouped in five categories: coupling, cohesion, inheritance, complexity and size. The objective was to find in which proportions the Qi metric captures the information provided by the selected OOM. The obtained results provided evidence that the Qi metric captures, overall, more than 75 % of the information provided by most of the evaluated metrics. The purpose of the present paper is to explore empirically the relationship between the Qi metric and testability of classes in OOS in terms of the effort needed for testing. We investigate testability from the perspective of unit testing, where units consist of the classes of an OOS. 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, we used different metrics to measure some characteristics of the corresponding JUnit test cases. In order to evaluate the capability of the new metric to predict testability of classes, we used statistical tests using correlation.

The remainder of the article is organized as follows: Section 2 gives a survey on related work on software testability. The proposed metric is presented in Section 3. In Section 4 we define the used metrics to quantify JUnit test classes, describe the experimental design and discuss the statistical technique we used. 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.

II. SOFTWARE TESTABILITY

IEEE [18] 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 [19] defines testability (characteristic of maintainability) as attributes of software that bear on the effort needed to validate the software product. Fenton et *al.* [11] 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 parameters as stated by Baudry et *al.* [3, 4]. Yeh et *al.* [35] argue also that diverse factors such as control flow, data flow, complexity and size contribute to testability. According to Zhao [36], testability is an elusive concept, and it is difficult to get a clear view on all the potential factors that can affect it. Freedman introduces testability measures for software components based on two factors: observability and controllability [12]. Voas defines testability as the probability that a test case will fail if the program has a fault [32]. Voas and Miller [33] 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 [34]. Binder [7] 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 [23, 24]. McGregor et *al.* [28] investigate testability of OOS and introduce the visibility component measure (VC). Bertolino et *al.* [6] investigate testability and its use in dependability assessment. Le Traon et *al.* [25, 26, 27] propose testability measures for data flow designs. Petrenko et *al.* [30] and Karoui et *al.* [21] address testability in the context of communication software. Sheppard et *al.* [31] focus on formal foundation of testability metrics. Jungmayr [20] investigates testability measurement based on static dependencies within OOS. Gao et *al.* [13] consider testability from the perspective of component-based software construction, and address component testability issues by introducing a model for component testability analysis [14]. Nguyen et *al.* [29] focus on testability analysis based on data flow designs in the context of embedded software. Baudry et *al.* address testability measurement (and improvement) of OO designs [3, 4, 5]. Metrics can, in fact, be used to locate parts of a program which contribute to a lack of testability. Bruntink et *al.* [8, 9] 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. Bruntink et *al.* investigate testability from the perspective of unit testing. Khan et *al.* [22] focus also on class level testability using OOM. Chowdhary [10] focuses on why it is so difficult to practice testability in the real world.

III. QUALITY ASSURANCE INDICATOR

In this section, we give a summary of the definition of the *Quality Assurance Indicator (Qi)* metric. The Qi metric is based on *control call graphs*, which are a reduced form of traditional *control flow graphs*. A *control call graph* is a *control flow graph* from which the nodes representing instructions, or basic blocs of sequential instructions, not containing a call to a method are removed. The Qi metric is normalized and gives values in the interval [0, 1]. A low value of the Qi of a class reflects that the class (is a high-risk class and) needs more testing effort to ensure its quality, while a high value indicates that (the class is a low-risk class knowing that) the testing effort invested effectively on the class is high (proportional to its complexity). The Qi of a class depends on the Qi of its collaborating classes (invoked classes).

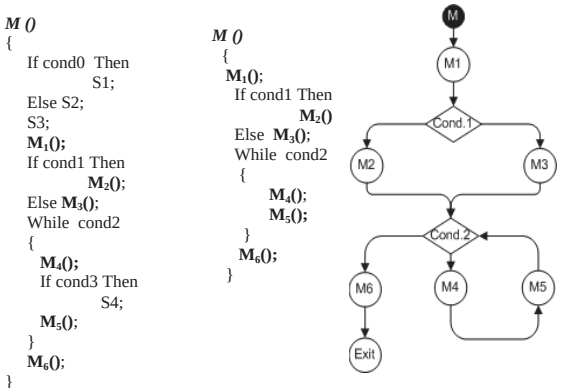

Figure 1 A method and its corresponding control call graph.

A. Control Call Graphs

Let us consider the example of method M given in Figure 1.1. The S_i represent blocs of instructions that do not contain a call to a method. The code of method M reduced to *control call flow* is given in Figure 1.2. The instructions (blocs of instructions) not containing a call to a method are removed from the original code of method M. Figure 1.3 gives the corresponding *control call graph*. Unlike traditional *call graphs*, *control call graphs* are much more precise models. They capture the structure of calls and related control.

B. Quality Assurance Indicator

We define the Qi of a method M_i as a kind of estimation of the probability that the control flow will go through the method without any *failure*. It may be considered as an indicator of the risk associated to a method (and a class at a high level). The Qi of a method M_i depends, in fact, on various intrinsic characteristics of the method itself, such as its unit

testing coverage (testing effort actually invested on the method) and its cyclomatic complexity, as well as on the Qi of all the methods invoked by the method M_i. We assume, in fact, that the quality of a method, particularly in terms of reliability, depends also on the quality of the methods it collaborates with to perform its task. In OOS, objects collaborate to achieve their respective responsibilities. A method of poor quality (lowly tested) can have (directly or indirectly) a negative impact on the methods that use it. There is here a kind of propagation, depending on the distribution of the control flow in a system, that needs to be captured. It is not obvious, particularly in the case of large and complex OOS, to identify intuitively this type of interferences between classes. The Qi of a method M_i is given by:

$$Qi_{M_i} = Qi^*_{M_i} * \sum_{j=1}^{n_i} \left(P\left(C_j^i\right) * \prod_{k \in \sigma} Qi_{M_k} \right) \quad (1)$$

with : Qi_{Mi}: quality assurance indicator of M_i, Qi^*_{Mi}: intrinsic quality assurance indicator of M_i, $P\left(C_j^i\right)$: probability of execution of path C_j^i of M_i, Qi_{Mk}: quality assurance indicator of the methods included in the path C_j^i, n_i: number of linear paths of the control call graph of M_i, and $Card(\sigma)=m_j$: number of the methods included in the path C_j^i. By applying the previous formula (1) to each method we obtain a system of N equations (N is the number of methods in the program). The obtained system is not linear and is composed of several multivariate polynomials. We use an iterative method (method of successive approximations) to solve it. The system is, in fact, reduced to a fixed point problem. Furthermore, we define the Qi of a class as the product of the Qi of its public methods. The calculation of the Qi metric is entirely automated by an Eclipse plug-in that we developed for Java software systems.

C. Assigning Probabilities

The *control call graph* of a method can be seen as a set of paths that the control flow can pass through. Passing through a particular path depends, in fact, on the states of the conditions in the control structures. To capture this probabilistic characteristic of the control flow, we assign a probability to each path C_k of a *control call graph* as follows:

$$P(C_k) = \prod_{i=0}^{n_k} P(A_i) \quad (2)$$

where A_i are the directed arcs composing the path C_k. By supposing, to simplify analysis and calculations, that the conditions in the control structures are independent, the $P(A_i)$ becomes the probability of an arc of being taken when exiting a control structure. The $P(C_k)$ are then reduced to a product of the probabilities of the states of the conditions in the control structures. To facilitate our experiments, we assigned probabilities to the different control structures of a Java program according to the rules given in TABLE I. These values are assigned automatically during the static analysis of the source code of a program when generating the Qi models. As an alternative way, the probabilities values would also be obtained by dynamic analysis, or assigned by programmers (knowing the code). Dynamic analysis will be considered in a future work.

TABLE I ASSIGNMENT RULES OF THE PROBABILITIES.

Nodes	Probability Assignment Rule
(if, else)	0.5 for the exiting arc « condition = true » 0.5 for the exiting arc « condition=false »
while	0.75 for the exiting arc « condition = true » 0.25 for the exiting arc « condition = false »
(do, while)	1 for the arc: (the internal instructions are executed at least once)
(switch,case)	1/n for each arc of the n cases.
(?, :)	0.5 for the exiting arc « condition = true » 0.5 for the exiting arc « condition = false »
for	1 for the arc
(try, catch)	0.75 for the arc of the « try » bloc 0.25 for the arc of the « catch » bloc
Polymorphism	1/n for each of the eventual n calls.

D. Intrinsic Quality Assurance Indicator

The *Intrinsic Quality Assurance Indicator* of a method M_i, noted Qi^*_{Mi}, depends on its cyclomatic complexity as well as on its unit testing coverage (as an indicator of the testing effort). It is given by:

$$Qi^*_{M_i} = \left(1 - \frac{F_i}{F_{\max}}\right) \quad (3)$$

with: $F_i = cc_i * (1 - tc_i)$, where : cc_i : cyclomatic complexity of the method M_i, and tc_i : unit testing coverage of the method M_i, $tc_i \in [0, 1]$.

$$F_{\max} = \max_{1 \le i \le N} \left(F_i\right)$$

Cyclomatic complexity can help software engineers determining the inherent risk of a program. Several studies provided empirical evidence that there is a significant relationship between cyclomatic complexity and fault proneness [1, 37]. Cyclomatic complexity is also recognized as a good indicator of testability [8, 9]. The more the cyclomatic complexity of a program is high, the more likely its testing effort would be high. Testing activities will reduce the risk of a complex program and achieve its quality. Moreover, testing coverage provide objective measures on the effectiveness of a testing process.

IV. EXPERIMENTAL DESIGN

The goal of this study is to explore empirically the relationship between the Qi metric and testability of classes in OOS. We evaluate the Qi metric at the class level, and limit the testing effort to the unit testing of classes. For our experiments, we selected from each of the used systems only the classes for which JUnit test cases exist. 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.

A. Metrics Related to Testability

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

1 www.junit.org

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. We used in our experiments each pair $<C_s, C_t>$ for classes for which test cases exist. The objective is to use these pairs to evaluate the relationship between the Qi metric and the measured characteristics of the test classes C_t. To capture testability of classes, we decided to measure for each test class C_t, corresponding to a software class C_s, various characteristics. We used the following suite of *test case metrics*:

- *TLOC*: This metric gives the number of lines of code of the test class C_t. This metric is used to indicate the size of the test suite corresponding to a class C_s.
- *TAssert*: This 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. It is directly related to the construction of the test cases.
- *THEff*: This measure is one of the Halstead Software Science metrics [16]. This metric gives the effort necessary to implement or understand a test class C_t. It is proportional to the volume and to the difficulty level of the test class. We assume that this will reflect also the difficulty of the class under test and the effort required to construct the corresponding test class.

The approach used in this paper is based on the work of Bruntik et *al.* [8]. Two of the used test case metrics (*TLOC* and *TAssert*) have, in fact, been introduced by Bruntink et *al.* in [8, 9] to indicate the size of a test suite. Bruntink et *al.* based the definition of these metrics on the work of Binder [7]. We assume, in this paper, that these metrics are indicators of the testability of classes. These metrics reflect different source code factors as stated by Bruntink et *al.* in [8, 9]: 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*. These two categories have been referred as *test case generation factors* and *test case construction factors*. 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 local variables or invoked methods). Since our work is exploratory in nature, we decided to extend the two metrics *TLOC* and *TAssert* by using the *THEff* metric to quantify the global effort necessary to implement a test class. We used this suite of metrics for characterizing the testing effort of classes. 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.

B. Data Collection

We calculated the values of the Qi (and Qi^*) metric for all classes for which JUnit test cases exist. We used the Eclipse plug-in we developed. For our experiments, we fixed the unit testing coverage to 75% for each of the methods of the analyzed systems. Furthermore, we evaluated other values of testing coverage. The obtained results were substantially the same (in a relative way). We also used the suite of test case metrics to quantify, for each of the subject systems, the JUnit test classes C_t that have been developed by the programmers using the JUnit framework. The test case metrics have been computed using the Borland Together tool.

C. Goal, Hypotheses and Statistical Analysis

We present, in this section, the methodology of the empirical study we conducted in order to assess the relationship between the Qi (and Qi*) metric and testability of 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 Qi metric and testability.
- H_1: There is a significant correlation between the Qi metric and testability.

In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the Qi metric and test case metrics (the chosen significance level is α=0.05). For the analysis of the collected data, we preferred a non-parametric measure of correlation in order to test the correlation between the Qi (and Qi*) metric and the suite of test case metrics. 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.

V. EMPIRICAL STUDY

A. 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.

TABLE II summarizes some characteristics of the analyzed systems : number of software classes, number of attributes, number of methods, number of lines of code, average value of lines of code, average value of cyclomatic complexity, percentage of tested classes (software classes for which JUnit test cases have been developed), number of JUnit test classes, and for software classes for which JUnit test classes have been developed : total number of lines of code, average value of lines of code and average value of cyclomatic complexity.

TABLE II SOME CHARACTERISTICS OF THE USED SYSTEMS.

	SOFTWARE CLASSES							TESTED SOFTWARE CLASSES			
SYSTEMS	#Classes	#Attributes	#Methods	LOC	MLOC	MWMPC	%TestedClasses	#TClasses	LOC	MLOC	MWMPC
ANT	713	2491	5365	64062	89.85	17.1	16.1%	115	17655	153.52	30.37
JFC	496	1550	5763	68312	137.73	28.09	46.37%	230	53131	231	46.08

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

B. Results

For each pair <C_s, C_t> we analyzed the collected data set by calculating the Spearman's correlation coefficient r_s for each pair of metrics. TABLE III summarizes the results of the correlation analysis. It shows, for each of the subject systems and between each distinct pair of metrics the obtained values for the Spearman's correlation coefficient. We also calculated the Spearman's correlation coefficient r_s for each pair of test case metrics (TABLE IV). The obtained Spearman's correlation coefficients that are significant (at α=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.

TABLE III CORRELATION VALUES BETWEEN THE Q_i AND Q_i* METRICS AND TEST CASE METRICS.

ANT	TLOC	TAssert	THEff
Qi	**-0.560**	**-0.326**	**-0.448**
Qi*	**-0.586**	**-0.393**	**-0.523**

JFC	TLOC	TAssert	THEff
Qi	**-0.425**	**-0.365**	**-0.353**
Qi*	**-0.447**	**-0.458**	**-0.439**

TABLE IV CORRELATION VALUES BETWEEN TEST CASE METRICS.

ANT	TLOC	TAssert	THEff
TLOC	1	**0.679**	**0.900**
TAssert		1	**0.781**
THEff			1

JFC	TLOC	TAssert	THEff
TLOC	1	**0.848**	**0.922**
TAssert		1	**0.896**
THEff			1

The first global observation that we can make is that the obtained results confirm that there is a significant relationship between the Qi and Qi* metrics and the used test case metrics. The obtained Spearman's correlation coefficients between the Qi and Qi* metrics and the test case metrics are all significant (at α=0.05) for the two selected systems (for all the pairs of metrics). We can reject the hypothesis H_0 and accept the hypothesis H_1. Moreover, the measures have negative correlation. As mentioned previously, a negative correlation indicates that the ranks of one variable (Qi and Qi* values in our case) decrease as the ranks of the other variable (test case metric) increase. These results are plausible knowing that the more classes (and methods) are complex, the more they are difficult to test and their Qi (and Qi*) values decrease. The second global observation that we can make is that the Qi* metric is, overall, better correlated to the test case metrics than the Qi metric. This may be explained by the fact that the metric Qi* takes into account only the inherent characteristics of methods (and classes) compared to the metric Qi which take into account the dependencies between methods (and classes). The other global observation that we can make is that the test case metrics are also correlated between themselves (TABLE IV).

C. Limitations

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 metrics Qi and Qi* and testability. In fact, there are a number of limitations that may affect the results of the study or limit their interpretation and generalization. The obtained results are based on the data set we collected from the analyzed systems. To collect data we only used a subset of classes (and corresponding JUnit test cases) from each of the subject systems. From TABLE II we can see that for system ANT we used only 115 software classes and corresponding JUnit test cases, and for system JFREECHART we used 230 software classes and corresponding JUnit test cases. In total, we analyzed 345 software classes and corresponding JUnit test classes. Even if we believe that the analyzed set of data is enough large to allow obtaining significant results, the study should be, however, replicated on a large number of OOS to increase the generality of the results. Moreover, we can also observe from TABLE II that the classes for which JUnit test cases have been developed are relatively large and complex classes. 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. By analyzing the source code of the 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. 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.

VI. CONCLUSIONS AND FUTURE WORK

We presented, in this paper, a metric capturing in an integrated way different attributes of OOS. The metric, called *Quality Assurance Indicator*, uses control flow paths and probabilities, and captures the collaboration between classes. The paper investigated empirically the relationship between the proposed metric and testability of classes. Testability has

been investigated from the perspective of unit testing. As a first attempt, we designed and performed an empirical study on two open source Java software systems, for which JUnit test cases exist. We used three metrics for characterizing the JUnit test classes. We performed statistical tests using correlation. The achieved results support the idea that there is a statistically and practically significant relationship between the Qi and Qi* metrics 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. We hope, however, this study will contribute to a better understanding and characterizing of what contributes to testability of classes in OOS. As future work, we plan: to extend the used test case metrics to better reflect the testing effort, to extend the study by using some well-known OOM, and to replicate the study on other projects to be able to give generalized results.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Aggarwal, K.K., Yogesh, S., Arvinder, K., and Ruchika, M., "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness": A replicated case study, Software Process: Improvement and Practice, 16 (1), 2009.

[2] Badri, M., Badri, L., Toure, F., "Empirical Analysis of Object-Oriented Design Metrics : Towards a new metric using control flow paths and probabilities", JOT, vol. 8(6), 2009.

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

[4] 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.

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

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

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

[8] Bruntink, M., Deursen, A.V., "Predicting Class Testability using Object-Oriented Metrics", 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE, 2004.

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

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

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

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

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

[14] Gao, J., Shih, M.C., "A Component Testability Model for Verification and Measurement", COMPSAC, IEEE, 2005.

[15] 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.

[16] Halstead, M. H., "Elements of Software Science", Elsevier/North-Holland, NY, 1977.

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

[18] IEEE, 1990. IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press, NY, 1990.

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

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

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

[22] Khan, R.A., Mustafa, K., "Metric Based Testability Model for Object-Oriented Design (MTMOOD)", ACM SIGSOFT Software Engineering Notes, vol. 34, no. 2, March 2009.

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

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

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

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

[27] Le Traon, Y., Ouabdessalam, F., Robach, C., "Analyzing testability on data flow designs", ISSRE'00, San Jose, 2000.

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

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

[30] Petrenko, A., Dssouli, R., and Koenig, H., "On Evaluation of Testability of Protocol Structures", IFIP, Pau, France, 1993.

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

[32] Voas, J.M., PIE: "A dynamic failure-based technique", IEEE TSE, 18(8), August 1992.

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

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

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

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

[37] Zhou, Y., Leung, H., "Empirical analysis of object-oriented design metrics for predicting high and low severity faults", IEEE Trans. on software engineering, vol. 32, no. 10, 2006.