Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 964064, 13 pages
doi:10.1155/2012/964064

Research Article

Evaluating the Effect of Control Flow on the Unit Testing Effort of Classes: An Empirical Analysis

Mourad Badri and Fadel Toure

Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec at Trois-Rivières, Trois-Rivières, QC, Canada G9A 5H7

Correspondence should be addressed to Mourad Badri, mourad.badri@uqtr.ca

Received 25 November 2011; Revised 18 March 2012; Accepted 28 March 2012

Academic Editor: Filippo Lanubile

Copyright © 2012 M. Badri and F. Toure. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The aim of this paper is to evaluate empirically the relationship between a new metric (*Quality Assurance Indicator*—Qi) and testability of classes in object-oriented systems. The Qi metric captures the distribution of the control flow in a system. We addressed testability from the perspective of unit testing effort. We collected data from five open source Java software systems for which JUnit test cases exist. To capture the testing effort of classes, we used different metrics to quantify the corresponding JUnit test cases. Classes were classified, according to the required testing effort, in two categories: high and low. In order to evaluate the capability of the Qi metric to predict testability of classes, we used the univariate logistic regression method. The performance of the predicted model was evaluated using Receiver Operating Characteristic (ROC) analysis. The results indicate that the univariate model based on the Qi metric is able to accurately predict the unit testing effort of classes.

1. Introduction

Software testing plays a crucial role in software quality assurance. It has, indeed, an important effect on the overall quality of the final product. Software testing is, however, a time and resources consuming process. The overall effort spent on testing depends, in fact, on many different factors including [1–5] human factors, process issues, testing techniques, tools used, and characteristics of the software development artifacts.

Software testability is an important software quality attribute. IEEE [6] 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 [7] defines testability (characteristic of maintainability) as attributes of software that bear on the effort needed to validate the software product. Dealing with software testability raises, in fact, several questions such as [8, 9]: Why is one class easier to test than another? What makes a class hard to test? What contributes to the testability of a class? How can we quantify this notion?

Metrics (or models based on metrics) can be used to predict (assess) software testability and better manage the testing effort. Having quantitative data on the testability of a software can, in fact, be used to guide the decision-making of software development managers seeking to produce high-quality software. Particularly, it can help software managers, developers, and testers to [8, 9] 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. One effective way to deal with this important issue is to develop prediction models that can be used to identify critical parts of the code requiring a (relative) high testing effort. Moreover, having quantitative data on the testing effort actually applied during the testing process (such as testing coverage measures) will also help to better identify, in an iterative and relative way, the critical parts of the code on which more testing effort is required to ensure software quality.

A large number of object-oriented (OO) metrics were proposed in the literature [10]. Some of these metrics, related to different OO attributes (such as size, complexity, coupling,

and cohesion), were already used in recent years to assess testability of OO software systems (e.g., [4, 8, 9, 11–15]). According to Gupta et al. [9], none of the OO metrics is alone sufficient to give an overall estimation of software testability. Software testability is, in fact, affected by many different factors [1–3, 5]. Moreover, few empirical studies have been conducted to examine the effect of these metrics on testability of classes, particularly when taking into account the testing effort level. As far as we know, this issue has not been empirically investigated. In addition, as mentioned by Baudry et al. [2, 3], testability becomes crucial in the case of OO software systems where control flows are generally not hierarchical but diffuse and distributed over whole architecture.

We proposed in [16] a new metric, called *Quality Assurance Indicator* (Qi), capturing in an integrated way different attributes of OO software systems such as complexity (control flow paths) and coupling (interactions between classes). The metric captures the distribution of the control flow in a system. The *Quality Assurance Indicator* of a class is based on different intrinsic characteristics of the class, as well as on the *Quality Assurance Indicator* of its collaborating classes (invoked classes). The metric has, however, no ambition to capture the overall quality (or testability) of OO software systems. Moreover, the objective is not to evaluate a design by giving absolute values, but more relative values that may be used for identifying: (1) before the testing process begins, the critical classes that will require a (relative) high testing effort, and (2) during the testing process, the classes on which more testing effort is required to ensure software quality (iterative distribution of the testing effort). In this paper, we focus on the first objective. Applying equal testing effort to all classes of a software is, indeed, cost-prohibitive and not realistic, particularly in the case of large and complex software systems. Increasing size and complexity of software systems brings, in fact, new research challenges. One of the most important challenges is to make testing effective with reasonable consumption of resources. We compared in [16] the Qi metric using the Principal Components Analysis (PCA) method to some well-known OO metrics. The evaluated metrics were grouped in five categories: coupling, cohesion, inheritance, complexity, and size. The achieved results provide evidence that the Qi metric captures, overall, a large part of the information captured by most of the evaluated metrics. Recently, we explored the relationship between the Qi metric and testability of classes [15]. Testability was basically measured (inversely) by the number of lines of test code and the number of *assert* statements in the test code. The relationship between the Qi metric and testability of classes was explored using only correlation analysis. Moreover, we have not distinguished among classes according to the required testing effort.

The purpose of the present paper is to evaluate empirically the relationship between the Qi metric and testability of classes in terms of required unit testing effort. The question we attempt to answer is how accurately do the Qi metric predicts (high) testing effort of classes. We addressed testability from the perspective of unit testing. We performed an empirical analysis using data collected from five 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. Classes were classified, according to the required unit testing effort, in two categories: high and (relatively) low. In order to evaluate the relationship between the Qi metric and testability of classes, we performed a statistical analysis using correlation and logistic regression. We used particularly the univariate logistic regression analysis to evaluate the effect of the Qi metric on the unit testing effort of classes. The performance of the predicted model was evaluated using Receiver Operating Characteristic (ROC) analysis. We also include in our study the well-known SLOC metric as a "baseline" to compare against the Qi metric (We wish to thank an anonymous reviewer for making this suggestion.). This metric is, indeed, one of the most used predictors in source code analysis. In summary, the results indicate that the Qi metric is a significant predictor of the unit testing effort of classes.

The rest of this paper is organized as follows: Section 2 gives a survey on related work on software testability. The Qi metric is introduced in Section 3. Section 4 presents the selected systems, describes the data collection, introduces the test case metrics we used to quantify the JUnit test cases, and presents the empirical study we performed to evaluate the relationship between the Qi metric and testability of classes. Finally, Section 5 summarizes the contributions of this work and outlines directions for future work.

2. Software Testability

Fenton and Pfleeger [17] define software testability as an external attribute. According to Gao and Shih [18], software testability is related to testing effort reduction and software quality. For Sheppard and Kaufman [19], software testability impacts test costs and provides a means of making design decisions based on the impact on test costs. Zhao [5] 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. According to Baudry et al. [2, 3], software testability is influenced by different factors including controllability, observability, and the global test cost. Yeh and Lin [1] argue also that diverse factors such as control flow, data flow, complexity, and size contribute to testability. Zhao [5] 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. Many testability analysis and measurement approaches have been proposed in the literature. These approaches were investigated within different application domains.

Freedman [20] introduces testability measures for software components based on two factors: observability and controllability. Observability is defined as the ease of determining if specific inputs affect the outputs of a component, and controllability is defined as the ease of producing specific outputs from specific inputs. Voas [21] defines testability as the probability that a test case will fail if a program has a fault. He considers testability as the combination of the probability that a location is executed, the probability of a fault at

a location, and the probability that corrupted results will propagate to observable outputs. Voas and Miller [22] propose a testability metric based on inputs and outputs domains of a software component, and the PIE (Propagation, Infection and Execution) technique to analyze software testability [23].

Binder [24] defines testability as the relative ease and expense of revealing software faults. He argues that software testability is based on six factors: representation, implementation, built-in text, test suite, test support environment, and software process capability. Each factor is further refined to address special features of OO software systems, such as inheritance, encapsulation, and polymorphism. Khoshgoftaar et al. [25] address the relationship between static software product measures and testability. Software testability is considered as a probability predicting whether tests will detect a fault. Khoshgoftaar et al. [26] use neural networks to predict testability from static software metrics.

McGregor and Srinivas [27] investigate testability of OO software systems and introduce the visibility component measure (VC). Bertolino and Strigini [28] investigate testability and its use in dependability assessment. They adopt a definition of testability as a conditional probability, different from the one proposed by Voas et al. [21], and derive the probability of program correctness using a Bayesian inference procedure. Le Traon et al. [29–31] propose testability measures for data flow designs. Petrenko et al. [32] and Karoui and Dssouli [33] address testability in the context of communication software. Sheppard and Kaufman [19] focus on formal foundation of testability metrics. Jungmayr [34] investigates testability measurement based on static dependencies within OO systems by considering an integration testing point of view.

Gao et al. [35] consider testability from the perspective of component-based software development and address component testability issues by introducing a model for component testability analysis [18]. The definition of component testability is based on five factors: understandability, observability, controllability, traceability, and testing support capability. According to Gao and Shih [18], software testability is not only a measure of the effectiveness of a testing process, but also a measurable indicator of the quality of a software development process. Nguyen et al. [36] focus on testability analysis based on data flow designs in the context of embedded software.

Baudry et al. [2, 3, 37] address testability measurement (and improvement) of OO designs. They focus on design patterns as coherent subsets in the architecture, and explain how their use can provide a way for limiting the severity of testability weaknesses. The approach supports the detection of undesirable configurations in UML class diagrams. Chowdhary [38] focuses on why it is so difficult to practice testability in the real world and discusses the impact of testability on design. Khan and Mustafa [39] focus on testability at the design level and propose a model predicting testability of classes from UML class diagrams. Kout et al. [40] adapt this model to the code level and evaluate it using two case studies.

Bruntink and Van Deursen [4, 8] investigate factors of testability of OO software systems using an adapted version of the *fish bone* diagram developed by Binder [24]. They studied five open source Java systems, for which JUnit test cases exist, in order to explore the relationship between OO design metrics and some characteristics of JUnit test classes. Testability is measured (inversely) by the number of lines of test code and the number of *assert* statements in the test code. This paper explores the relationship between OO metrics and testability of classes using only correlation analysis and did not distinguish among classes according to the testing effort.

Singh et al. [11] use OO metrics and neural networks to predict the testing effort. The testing effort in this work is measured in terms of lines of code added or changed during the life cycle of a defect. Singh et al. conclude that the performance of the developed model is to a large degree dependent on the data used. In [41], Singh and Saha attempt to predict the testability of Eclipse at package level. This study was, however, limited to a correlation analysis between source code metrics and test metrics. Badri et al. [13] performed a similar study to that conducted by Bruntink and Van Deursen [4, 8] using two open source Java systems in order to explore the relationship between lack of cohesion metrics and testability characteristics. In [14], Badri et al. investigate the capability of lack of cohesion metrics to predict testability of classes using logistic regression methods.

3. 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 the concept of *Control Call Graphs*, which are a reduced form of traditional *Control Flow Graphs*. A control call graph is, in fact, 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 means that the class is a high-risk class and needs a (relative) high testing effort to ensure its quality. A high value of the Qi of a class indicates that the class is a low-risk class (having a relatively low complexity and/or the testing effort applied actually on the class is relatively high—proportional to its complexity).

3.1. Control Call Graphs. Let us consider the example of method M given in Figure 1(a). 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(b). The instructions (blocs of instructions) not containing a call to a method are removed from the original code of method M. Figure 1(c) 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.

3.2. 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 with a method (and a class at a high level). The Qi of a method M_i is based, in fact, on intrinsic characteristics of

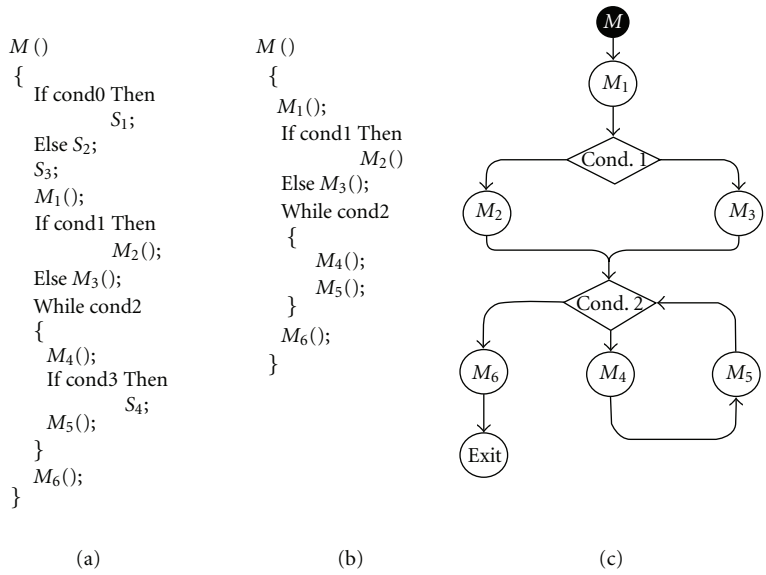


Figure 1: A method and its corresponding control call graph.

the method, such as its cyclomatic complexity and its unit testing coverage (testing effort applied actually on the method), as well as on the Qi of the methods invoked by the method M_i. We assume 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 OO software systems, objects collaborate to achieve their respective responsibilities. A method of poor quality 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 OO software systems, to identify intuitively this type of interferences between classes. This type of information is not captured by traditional OO metrics. The Qi of a method M_i is given by:

$$\mathrm{Qi}_{M_i} = \mathrm{Qi}^*_{M_i} \cdot \sum_{j=1}^{n_i} \left[P\left(C_j^i\right) \cdot \prod_{M \in \sigma_j} \mathrm{Qi}_M \right] \tag{1}$$

with Qi_{M_i}: quality assurance indicator of method M_i, $\mathrm{Qi}^*_{M_i}$: intrinsic quality assurance indicator of method M_i, C_j^i: jth path of method M_i, $P(C_j^i)$: probability of execution of path C_j^i of method M_i, Qi_M: quality assurance indicator of the method M included in the path C_j^i, n_i: number of linear paths of the control call graph of method M_i, and σ_j: set of the methods invoked 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. In order to better understand our approach, we give in what follows the Qi of the method M given in Figure 1 as a simple example of application:

$$\begin{aligned}\mathrm{Qi}_M = \mathrm{Qi}^*_M \Big[& \mathrm{Qi}_{M_1} \left(0.5\mathrm{Qi}_{M_2} + 0.5\mathrm{Qi}_{M_3}\right) \\ & \times \left(0.75\mathrm{Qi}_{M_4}\mathrm{Qi}_{M_5}\mathrm{Qi}_{M_6} + 0.25\mathrm{Qi}_{M_6}\right)\Big].\end{aligned} \tag{2}$$

Furthermore, we define the Qi of a class C (noted Qi_C) as the product of the Qi of its methods:

$$\mathrm{Qi}_C = \prod_{M \in \delta} \mathrm{Qi}_{M,} \tag{3}$$

where δ is the set of methods of the class C. The calculation of the Qi metric is entirely automated by a tool that we developed for Java software systems.

3.3. 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 of a control call graph as follows:

$$P(C) = \prod_{A \in \theta} P(A), \tag{4}$$

where θ is the set of directed arcs composing the path C and $P(A)$ the probability of an arc to be crossed when exiting a control structure.

To facilitate our experiments (simplify analysis and calculations), we assigned probabilities to the different control structures of a Java program according to the rules given in Table 1. 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 probability values may also be assigned by programmers (knowing the code) or obtained by dynamic analysis. Dynamic analysis is out of the scope of this paper.

Table 1: 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	0.75 for entering the loop 0.25 for skipping the loop
(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

3.4. Intrinsic Quality Assurance Indicator. The *Intrinsic Quality Assurance Indicator* of a method M_i, noted $\mathrm{Qi}^*_{M_i}$, is given by

$$\mathrm{Qi}^*_{M_i} = (1 - F_i) \tag{5}$$

with

$$F_i = \frac{(1 - \mathrm{tc}_i)\mathrm{CC}_i}{\mathrm{CC}_{\max}}, \tag{6}$$

where CC_i: cyclomatic complexity of method M_i,

$$\mathrm{CC}_{\max} = \max_{1 \le i \le N} (\mathrm{CC}_i) \tag{7}$$

tc_i: unit testing coverage of the method M_i, $\mathrm{tc}_i \in [0, 1]$.

Studies provided empirical evidence that there is a significant relationship between cyclomatic complexity and fault proneness (e.g., [42–44]). 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.

4. Empirical Analysis

The goal of this study is to evaluate empirically the relationship between the Qi metric and testability of classes in terms of required testing effort. We selected from each of the investigated systems only the classes for which JUnit test cases exist. We noticed that developers usually name the JUnit test case classes by adding the prefix (suffix) "Test" ("TestCase") into the name of the classes for which JUnit test cases were developed. Only classes that have such name-matching mechanism with the test case class name are included in the analysis. This approach has already been adopted in other studies [45].

JUnit (http://www.junit.org/) is, in fact, a simple Framework for writing and running automated unit tests for Java classes. Test cases in JUnit are written by testers in Java. JUnit gives testers some support so that they can write those test cases more conveniently. A typical usage of JUnit is to test each class C_s of the program by means of a dedicated test case class C_t. To actually test a class C_s, we need to execute its test class C_t. This is done by calling JUnit's test runner tool. JUnit will report how many of the test methods in C_t succeed, and how many fail. However, we noticed by analyzing the JUnit test case classes of the subject systems that in some cases there is no one-to-one relationship between JUnit classes and tested classes. This has also been noted in other previous studies (e.g., [46, 47]). In these cases, several JUnit test cases have been related to a same tested class. The matching procedure has been performed on the subject systems by two research assistants separately (a Ph.D. student (second author of this paper) and a Master student, both in computer science). We compared the obtained results and noticed only a few differences. We rechecked the few results in which we observed differences and chose the correct ones based on our experience and a deep analysis of the code.

For each software class C_s selected, we calculated the value of the Qi metric. We also used the suite of test case metrics (Section 4.2) to quantify the corresponding JUnit test class (classes) C_t. The Qi metric has been computed using the tool we developed, and the test case metrics have been computed using the Borland (http://www.borland.com/) Together tool. For our experiments, knowing that the purpose of this study is to evaluate the relationship between the Qi metric and testability of classes, and that one of the main interests of such a work is to be able to predict the testing effort of classes using the Qi metric before the testing process begins, the testing coverage (tc_i, Section 3.4) is set to zero for all methods. As mentioned previously, we also include in our experiments the well-known SLOC (Source Lines Of Code) metric. The value of the SLOC metric has been computed, for each software class selected, using the Borland Together tool.

In this section, we present the systems we selected, discuss some of their characteristics, introduce the test case metrics we used to quantify the JUnit test cases, and present the empirical study we conducted to evaluate the relationship between the Qi metric and testability of classes in two steps: (1) analyzing correlations between the Qi metric and test case metrics and (2) evaluating the effect of the Qi metric on testability of classes, using univariate logistic regression, when the testing effort level is taken into account.

4.1. Selected Systems. Five open source Java software systems from different domains were selected for the study: ANT, JFREECHART (JFC), JODA-Time (JODA), Apache Commons IO (IO), and Apache Lucene Core (LUCENE). Table 2 summarizes some of their characteristics. It gives, for each system, the number of software classes, the number of attributes, the number of methods, the total number of lines of code, the number of selected software classes for which JUnit test cases were developed, and the total number of lines of code of selected software classes for which JUnit test cases were developed.

ANT (http://www.apache.org/) is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. This system consists of 713 classes that are

Table 2: Some characteristics of the used systems.

	No. classes	No. attributes	No. methods	No. LOC	No. TClasses	No. TLOC
ANT	713	2491	5365	64062	111 (15.6%)	17609 (27.5%)
JFC	496	1550	5763	68312	226 (45.6%)	53115 (77.8%)
JODA	225	872	3605	31591	76 (33.8%)	17624 (55.8%)
IO	104	278	793	7631	66 (63.5%)	6326 (82.9%)
LUCENE	659	1793	4397	56902	114 (17.3%)	22098 (38.8%)

comprised of 2491 attributes and 5365 methods, with a total of roughly 64000 lines of code. JFC (http://www.jfree.org/jfreechart/) is a free chart library for Java platform. This system consists of 496 classes that are comprised of 1550 attributes and 5763 methods, with a total of roughly 68000 lines of code. JODA-Time (Java date and time API) (http://joda-time.sourceforge.net/) is the de facto standard library for advanced date and time in Java. Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API. This system consists of 225 classes that are comprised of 872 attributes and 3605 methods, with a total of roughly 31000 lines of code. Apache Commons IO (IO) (http://commons.apache.org/io/) is a library of utilities to assist with developing Input-Output functionality. This system consists of 104 classes that are comprised of 278 attributes and 793 methods, with a total of roughly 7600 lines of code. LUCENE (Apache Lucene Core) (http://lucene.apache.org/) is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. This system consists of 659 classes that are comprised of 1793 attributes and 4397 methods, with a total of roughly 56900 lines of code.

We can also observe from Table 2, for each system, that JUnit test cases were not developed for all classes. The percentage of selected software classes for which JUnit test cases were developed varies from one system to another: (1) ANT: 111 classes, which represents 15.6% of the classes in the system. The total number of lines of code of these classes is 17609, which represents 27.5% of the total number of lines of code of ANT. (2) JFC: 226 classes, which represents 45.6% of the classes in the system. The total number of lines of code of these classes is 53115, which represents 77.8% of the total number of lines of code of JFC. (3) JODA: 76 classes, which represents 33.8% of the classes in the system. The total number of lines of code of these classes is 17624, which represents 55.8% of the total number of lines of code of JODA. (4) IO: 66 classes, which represents 63.5% of the classes in the system. The total number of lines of code of these classes is 6326, which represents 82.9% of the total number of lines of code of IO. (5) Finally, LUCENE: 114 classes, which represents 17.3% of the classes in the system. The total number of lines of code of these classes is 22098, which represents 38.8% of the total number of lines of code of LUCENE. So, in total, our experiments will be performed on 593 classes and corresponding JUnit test cases.

Moreover, the software classes for which JUnit test cases were developed, in the five subject systems, are relatively large and complex.

(i) For ANT, the mean values of their *lines of code* and *cyclomatic complexity* (resp., 158.64 and 31.31—Standard deviation (σ): 154.2 and 31.1) are greater than the mean values of the same measures for all classes (resp., 89.85 and 17.10—σ: 130.15 and 23.66). The same trend is observed for other systems.

(ii) For JFC, the mean values of their *lines of code* and *cyclomatic complexity* are, respectively, 235.02 and 46.89 (σ: 273.12 and 57.17) and the mean values of the same measures for all classes are, respectively, 137.73 and 28.10 (σ: 216.12 and 44.51).

(iii) For JODA, the mean values of their *lines of code* and *cyclomatic complexity* are, respectively, 231.90 and 44.75 (σ: 277.81 and 39.72) and the mean values of the same measures for all classes are, respectively, 140.40 and 28.74 (σ: 204.41 and 29.85).

(iv) For IO, the mean values of their *lines of code* and *cyclomatic complexity* are, respectively, 95.85 and 22.46 (σ: 143.25 and 37.53) and the mean values of the same measures for all classes are, respectively, 73.38 and 17.65 (σ: 119.95 and 31.24).

(v) Finally, for LUCENE, the mean values of their *lines of code* and *cyclomatic complexity* are, respectively, 193.84 and 35.89 (σ: 339.15 and 60.91) and the mean values of the same measures for all classes are, respectively, 86.35 and 16.64 (σ: 187.45 and 34.69).

4.2. Test Case Metrics. In order to indicate the testability of a software class (noted C_s), we used the following suite of test case metrics to quantify the corresponding JUnit test class (noted C_t).

(i) *TLoc*. This metric gives the number of lines of code of a test class C_t. It is used to indicate the size of the test suite corresponding to a software class C_s.

(ii) *TAss*. This metric gives the number of invocations of JUnit *assert* methods that occur in the code of a test class C_t. JUnit *assert* methods are, in fact, used by the testers to compare the expected behavior of the class under test to its current behavior. 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.

We used in our study the selected software classes and the corresponding JUnit test cases. The objective was to use these classes to evaluate the relationship between the Qi metric, which captures in an integrated way different characteristics of a software class C_s, and the measured characteristics of the corresponding JUnit test case (s). The approach used in this paper is, in fact, based on the work of Bruntink and Van Deursen [4, 8]. The test case metrics TLoc and TAss have been introduced by Bruntink and Van Deursen in [4, 8] to indicate the size of a test suite. Bruntink and Van Deursen based the definition of these metrics on the work of Binder [24]. They used, particularly, an adapted version of the *fish bone* diagram developed by Binder [24] to identify factors of testability. These metrics reflect different source code factors [4, 8]: 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* and *test case construction* factors.

However, by analyzing the source code of the JUnit test classes of the systems we selected for our study, we found that some characteristics of the test classes (which are also related to the factors mentioned above) are not captured by these two metrics (like the set of local variables or invoked methods). This is why we decided to extend these metrics. In [15], we used the *THEff* metric, which is one of the Halstead Software Science metrics [48]. The *THEff* metric gives the effort necessary to implement or understand a test class C_t. It is calculated as "Halstead Difficulty" $*$ "Halstead Program Volume." Halsteasd Program Volume is defined as: $N \log_2 n$, where N = Total Number of Operators + Total Number of Operands and n = Number of Distinct Operators + Number of Distinct Operands. In this work, we wanted to explore the *THDiff* metric. This metric is also one of the Halstead Software Science metrics [48]. It gives the difficulty level of a test class C_t. It is calculated as ("Number of Distinct Operators"/2) $*$ ("Total Number of Operands"/"Number of Distinct Operands"). We assume that this will reflect also the difficulty of the class under test and the global effort required to construct the corresponding test class.

In order to understand the underlying orthogonal dimensions captured by the test case metrics, we performed a Principal Component Analysis (PCA) using the four test case metrics (TLoc, TAss, THEff, and THDiff). PCA is a technique that has been widely used in software engineering to identify important underlying dimensions captured by a set of metrics. We used this technique to find whether the test case metrics are independent or are capturing the same underlying dimension (property) of the object being measured. The PCA was performed on the data set consisting of test case metrics values from JFC system. As it can be seen from Table 2, JFC is the system that has the most JUnit test cases.

The PCA identified two Principal Components (PCs), which capture more than 90% of the data set variance (Table 3). Based on the analysis of the coefficients associated with each metric within each of the components, the PCs are interpreted as follows: (1) PC_1: TLoc and TAss. These metrics are, in fact, size-related metrics. (2) PC_2: THDiff. This is the Halstead Difficulty measure. It captures more data variance than the THEff (Halstead Effort) measure. The results of the PCA analysis suggest that (1) the information provided by the metric THEff is captured by the (size related) test case metrics TLoc and TAss, and (2) the metric THDiff is rather complementary to the test case metrics TLoc and TAss. So, we used in this work the suite of metrics (TLoc, TAss, and THDiff) to quantify the JUnit test cases. 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.

Table 3: Results of PCA analysis.

	PC_1	PC_2	PC_3
Prop (%)	79.974	12.136	5.554
Cumul (%)	79.974	92.111	97.665
TAss	**26.168**	23.958	0.001
THDiff	23.51	**29.656**	46.749
THEff	24.112	22.848	**52.964**
TLoc	**26.209**	23.538	0,287

4.3. Correlation Analysis. In this section, we present the first step of the empirical study we performed to explore the relationship between the Qi metric and test case metrics. We performed statistical tests using correlation. We used a nonparametric 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 (http://www.xlstat.com/) tool to perform the statistical analysis.

As mentioned previously, we also include the SLOC metric in our experiments. So, we analyzed the collected data set by calculating the Spearman's correlation coefficient r_s for each pair of metrics (source code metric (Qi, SLOC) and test case metric). Table 4 summarizes the results of the correlation analysis. It shows, for each of the selected systems and between each distinct pair of metrics, the obtained values for the Spearman's correlation coefficient. The Spearman's correlation coefficients are all significant. The chosen significance level is $\alpha = 0.05$. In summary, as it can be seen from Table 4, the results confirm that there is a significant relationship (at the 95% confidence level) between the Qi and SLOC metrics and the used test case metrics for all the subject systems. Moreover, the observed correlation values between the source code metrics (Qi and SLOC) and the test case metrics are generally comparable.

Table 4: Correlation values between Qi and SLOC metrics and test case metrics.

	ANT			JFC			JODA			IO			LUCENE		
	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc
Qi	−0.361	−0.331	−0.553	−0.341	−0.209	−0.415	−0.762	−0.698	−0.805	−0.574	−0.550	−0.772	−0.467	−0.306	−0.457
SLOC	0.391	0.387	0.582	0.414	0.261	0.437	0.726	0.630	0.764	0.641	0.585	0.827	0.495	0.316	0.470

Table 5: Correlation values between test case metrics.

	ANT			JFC			JODA			IO			LUCENE		
	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc	TAss	THDiff	TLoc
TAss	1	0,73	0,77	1	0,83	0,84	1	0,90	0,95	1	0,60	0,79	1	0,59	0,77
THDiff		1	0,79		1	0,76		1	0,91		1	0,74		1	0,85
TLoc			1			1			1			1			1

The measures of correlations between the Qi metric and the test case metrics are negative. As mentioned previously, a negative correlation indicates that the ranks of one variable (Qi metric) decrease as the ranks of the other variable (test case metric) increase. These results are plausible and not surprising. Indeed, as mentioned in Section 3, a low value of the Qi of a class indicates that the class is a high-risk class and needs a high testing effort to ensure its quality. A high value of the Qi of a class indicates that the class is a low-risk class and needs a relatively low testing effort. The measures of correlations between the size-related SLOC metric and the test case metrics are positive. These results are, in fact, plausible. A large class, containing a large number of methods in particular, will require a high testing effort.

We also calculated the Spearman's correlation coefficient r_s for each pair of test case metrics (Table 5). The global observation that we can make is that the test case metrics are significantly correlated between themselves. The chosen significance level here also is $\alpha = 0.05$.

4.4. Evaluating the Effect of the Qi Metric on Testability Using Logistic Regression Analysis. In this section, we present the empirical study we conducted in order to evaluate the effect of the Qi metric on testability of classes in terms of testing effort. We used the univariate logistic regression analysis.

4.4.1. Dependent and Independent Variables. The binary dependent variable in our study is testability of classes. We consider testability from the perspective of unit testing effort. The goal is to evaluate empirically, using logistic regression analysis, the relationship between the Qi metric (independent variable in our study) and testability of classes. Here also, we used the SLOC metric as a "baseline" to compare against the Qi metric. We used the test case metrics (TLoc, TAss, and THDiff) to identify the classes which required a (relative) high testing effort (in terms of size and difficulty). As mentioned earlier, the metrics TLoc and TAss have been introduced by Bruntink and Van Deursen [4, 8] to indicate the size of a test suite. These metrics reflect, in fact, different source code factors [4, 8]: 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. In order to simplify the process of testing effort categorization, and as a first attempt, we provide in this study only two categorizations: classes which required a high testing effort and classes which required a (relative) low testing effort. In a first step, we used the three test case metrics to divide the test classes into four groups as follows.

Table 6: Distribution of classes.

	1	0
ANT	33.3%	66.7%
JFC	31.4%	68.6%
JODA	32.9%	67.1%
IO	30.3%	69.7%
LUCENE	29%	71%

Group 4. This group includes the JUnit test cases for which the three following conditions are satisfied: (1) large number of lines of code (corresponding TLoc ≥ mean value of TLoc), (2) large number of invocations of JUnit *assert* methods (corresponding TAss ≥ mean value of TAss), and (3) high difficulty level (corresponding THDiff ≥ mean value of THDiff).

Group 3. This group includes the JUnit test cases for which only two of the conditions mentioned above are satisfied.

Group 2. This group includes the JUnit test cases for which only one of the conditions mentioned above is satisfied.

Group 1. This group includes the JUnit test cases for which none of the conditions mentioned above is satisfied.

In a second step, we merged these four groups in two categories according to the testing effort as follows: high (groups 4 and 3) and low (groups 2 and 1). We affected the value 1 to the first category and the value 0 to the second one. Table 6 summarizes the distribution of classes according to the adopted categorization. From Table 6, it can be seen that for (1) ANT, 33.3% of the selected classes for which JUnit test cases were developed have been categorized as classes having required a high testing effort. (2) JFC, 31.4% of the selected classes for which JUnit test cases were developed have been

categorized as classes having required a high testing effort. (3) JODA, 32.9% of the selected classes for which JUnit test cases were developed have been categorized as classes having required a high testing effort. (4) IO, 30.3% of the selected classes for which JUnit test cases were developed have been categorized as classes having required a high testing effort. (5) Finally, for LUCENE, 29% of the selected classes for which JUnit test cases were developed have been categorized as classes having required a high testing effort. As it can be seen from Table 6, overall, one third of the classes (of each system) were categorized as classes having required a high testing effort.

4.4.2. Hypothesis. In order to evaluate the relationship between the Qi metric (and SLOC) and testability of classes, and particularly to find the effect of the Qi metric (and SLOC) on the testing effort, the study tested the following hypothesis.

Hypothesis 1 (Qi). A class with a low Qi value is more likely to require a high testing effort than a class with a high Qi value.

The Null Hypothesis. A class with a low Qi value is no more likely to require a high testing effort than a class with a high Qi value.

Hypothesis 2 (SLOC). A class with a high SLOC value is more likely to require a high testing effort than a class with a low SLOC value.

The Null Hypothesis. A class with a high SLOC value is no more likely to require a high testing effort than a class with a low SLOC value.

4.4.3. Logistic Regression Analysis: Research Methodology. Logistic Regression (LR) is a standard statistical modeling method in which the dependent variable can take on only one of two different values. It is suitable for building software quality classification models. It is used to predict the dependent variable from a set of independent variables and to determine the percent of variance in the dependent variable explained by the independent variables [42–44]. This technique has been widely applied to the prediction of fault-prone classes (e.g., [12, 43, 49–52]). LR is of two types: Univariate LR and Multivariate LR. A multivariate LR model is based on the following equation:

$$P(X_1, \ldots, X_n) = \frac{e^{(a+\sum_{i=1}^{i=n} b_i X_i)}}{1 + e^{(a+\sum_{i=1}^{i=n} b_i X_i)}}. \quad (8)$$

The X_is are the independent variables and the b_is are the estimated regression coefficients (approximated contribution) corresponding to the independent variables X_is. The larger the (normalized) absolute value of the coefficient, the stronger the impact of the independent variable on the probability of detecting a high testing effort. P is the probability of detecting a class with a high testing effort. The univariate regression analysis is a special case of the multivariate regression analysis, where there is only one independent variable (Qi or SLOC in our study).

The regression analysis here is not intended to be used to build a prediction model combining the two source code metrics (Qi and SLOC). Such models, and multivariate LR analysis, are out of the scope of this paper. Instead, our analysis intends to investigate the effect of the Qi metric on the testing effort and to compare it to the effect of the SLOC metric (taken as a well-known and proper baseline), in order to evaluate the actual benefits (ability) of the Qi metric when used to predict testability.

4.4.4. Model Evaluation. Precision and recall are traditional evaluation criteria that are used to evaluate the prediction accuracy of logistic regression models. Because precision and recall are subject to change as the selected threshold changes, we used the ROC (*Receiver Operating Characteristics*) analysis to evaluate the performance of the predicted model. The ROC curve, which is defined as a plot of sensitivity on the y-coordinate versus its 1-specificity on the x-coordinate, is an effective method of evaluating the quality (performance) of prediction models [53].

The ROC curve allows also obtaining a balance between the number of classes that the model predicts as requiring a high testing effort, and the number of classes that the model predicts as requiring a low testing effort. The optimal choice of the cut-off point that maximizes both sensitivity and specificity can be selected from the ROC curve. This will allow avoiding an arbitrary selection of the cut-off. In order to evaluate the performance of the prediction model, we used the AUC (Area Under the Curve) measure. It allows appreciating the model without subjective selection of the cutoff value. It is a combined measure of sensitivity and specificity. The lager the AUC measure, the better the model is at classifying classes. A perfect model that correctly classifies all classes has an AUC measure of 1. An AUC value close to 0.5 corresponds to a poor model. An AUC value greater than 0.7 corresponds to a good model [54].

Moreover, the issue of training and testing data sets is very important during the construction and evaluation of prediction models. If a prediction model is built on one data set (used as training set) and evaluated on the same data set (used as testing set), then the accuracy of the model will be artificially inflated [55]. A common way to obtain a more realistic assessment of the predictive ability of the model is to use cross validation (k-fold cross-validation), which is a procedure in which the data set is partitioned in k subsamples (groups of observation). The regression model is built using k−1 groups and its predictions evaluated on the last group. This process is repeated k times. Each time, a different subsample is used to evaluate the model, and the remaining subsamples are used as training data to build the model. We performed, in our study, a 10-fold cross-validation. We used the XLSTAT and R (http://www.r-project.org/) tools.

4.4.5. Univariate LR Analysis: Results and Discussion. Table 7 summarizes the results of the univariate LR analysis. The (normalized) *b-coefficient* is the estimated regression coefficient. The larger the absolute value of the coefficient, the stronger the impact of the Qi (SLOC) on the probability of detecting a high testing effort. The *P-value* (related to

Table 7: Results for univariate LR analysis.

		ANT	JFC	JODA	IO	LUCENE
Qi	R^2	0.365	0.160	0.214	0.488	0.177
	2Log	<0.0001	<0.0001	0.000	<0.0001	0.000
	b	−0.787	−0.478	−0.838	−1.066	−0.431
	P-value	<0.0001	<0.0001	0.022	<0.0001	0.000
	AUC	**0.81**	**0.72**	**0.85**	**0.89**	**0.70**
SLOC	R^2	0.255	0.194%	0.192	0.593	0.152
	2Log	<0.0001	<0.0001	0.001	<0.0001	0.000
	b	0.589	0.545	0.562	3.022	0.605
	P-value	<0.0001	<0.0001	0.006	0.000	0.011
	AUC	**0.80**	**0.75**	**0.80**	**0.91**	0.67

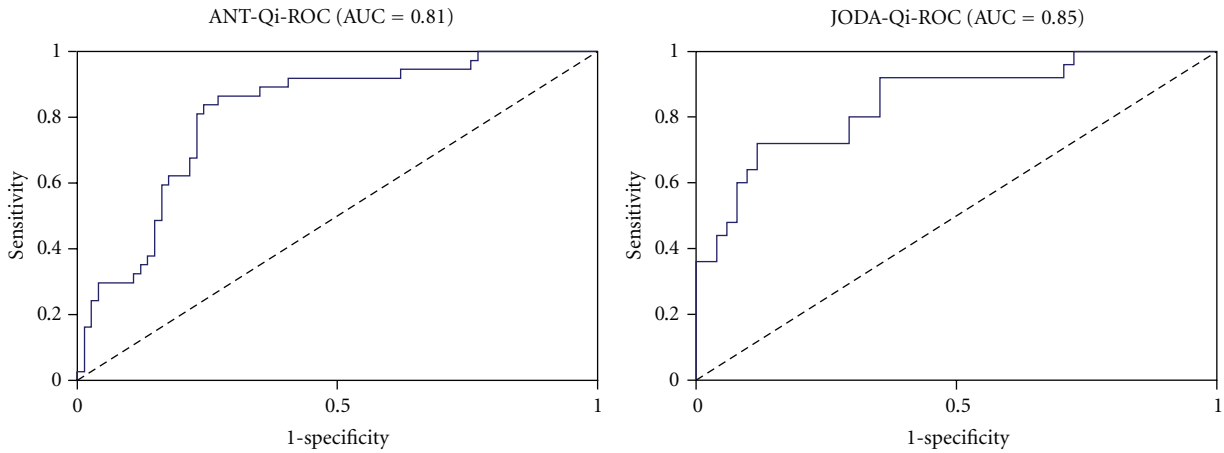


Figure 2: Univariate LR ROC curves for ANT and JODA systems.

the statistical hypothesis) is the probability of the coefficient being different from zero by chance and is also an indicator of the accuracy of the coefficient estimate. To decide whether the Qi and SLOC metrics are statistically significant predictors of testing effort, we used the $\alpha = 0.05$ significance level to assess the *P-value*. R^2 (Nagelkerke) is defined as the proportion of the total variance in the dependent variable (testing effort) that is explained by the model. The higher R^2 is, the higher is the effect of the Qi metric (and SLOC metric), and the more accurate is the model. In summary, the results show that, for the five investigated systems, the *b-coefficient* and the R^2 values of the Qi and SLOC metrics are significant. According to these results, we can conclude that the Qi and SLOC metrics are significantly related to the testing effort. The AUC values confirm that the univariate LR models based on the metrics Qi and SLOC are able to accurately predict the unit testing effort of classes.

The results show, for system ANT, for example, that the normalized b-coefficients of the metrics Qi and SLOC (resp., 0.787 and 0.589) are significantly different from zero according to their P-values. The used significance level is 0.05. The metric Qi has the highest R^2 value (0.365). According to the obtained results, the metrics Qi and SLOC are significantly related to the testing effort. The AUC values confirm that univariate LR models based on the metrics Qi and SLOC are able to accurately predict the unit testing effort of classes. However, we can see from Table 7 that the univariate LR model based on the metric Qi is slightly more predictive of testing effort than the one based on the metric SLOC (R^2 and *b-coefficient* values). Overall, the accuracies of both models are comparable, depending upon systems, except may be for system LUCENE where the model based on the SLOC metric has an AUC score of 0.67. For system LUCENE, the model based on the Qi metric has an AUC score of 0.7. Figure 2 gives the univariate LR ROC curves of the Qi metric for ANT and JODA systems.

4.5. Threats to Validity. 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 Qi metric and testability of classes. In fact, there are a number of limitations that may affect the results of the study or limit their interpretation and generalization.

The achieved results are based on the data set we collected from the investigated systems. As mentioned earlier (Section 4.1), we analyzed 593 Java classes and corresponding JUnit test cases. Even if we believe that the analyzed data

set is large enough to allow obtaining significant results, we do not claim that our results can be generalized to all systems. The study should be replicated on a large number of OO software systems to increase the generality of our findings.

Moreover, the classes for which JUnit test cases were developed, and this in all the investigated systems, are relatively large and complex. 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 allow observing the performance of the prediction models with data collected from classes of varying sizes (small, medium, and large).

It is also possible that facts such as the development style used by the developers for writing test cases and the criteria they used while selecting the software classes for which they developed test classes (randomly or depending on their size or complexity e.g., or on other criteria) may affect the results or produce different results for specific applications. We observed, in fact, that in some cases the developed JUnit classes do not cover all the methods of the corresponding software classes. This may be due to the style adopted by the developers while writing the test cases (or other considerations). As the source code metrics (Qi and SLOC) are computed using the complete code of the classes, this may affect (bias) the results.

Finally, another important threat to validity is from the identification of the relationship between the JUnit test cases and tested classes. As mentioned in Section 4, we noticed by analyzing the code of the JUnit test cases of the investigated systems that, in some cases, there is no one-to-one relationship between JUnit test cases and tested classes. In these cases, several JUnit test cases have been related to a same tested class. Even if we followed a systematic approach for associating the JUnit test cases to the corresponding tested classes, which was not an easy task, unfortunately we have not been able to do that for all classes. This may also affect the results of our study or produce different results from one system to another.

5. Conclusions and Future Work

The paper investigated empirically the relationship between a metric (Quality Assurance Indicator—Qi) that we proposed in a previous work and testability of classes in terms of required testing effort. The Qi metric captures, in an integrated way, different OO software attributes. Testability has been investigated from the perspective of unit testing. We performed an empirical analysis using data collected from five 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. Classes were classified according to the required testing effort in two categories: high and low. In order to evaluate the relationship between the Qi metric and testability of classes, we used the univariate logistic regression method. The performance of the predicted model was evaluated using Receiver Operating Characteristic (ROC) analysis. We also include in our study the well-known SLOC metric as a "baseline."

The results indicate that (1) the Qi metric is statistically related to the test case metrics and (2) the univariate regression model based on the Qi metric is able to accurately predict the unit testing effort of classes. Overall, the accuracies of the model based on the Qi metric and the one based on the SLOC metric are comparable. Based on these results, we can reasonably claim that the Qi metric is a significant predictor of the unit testing effort of classes. We hope these findings will help to a better understanding of what contributes to testability of classes in OO systems, and particularly the effect of control flow on the testing effort.

The performed study should, however, be replicated using many other OO software 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 extend the used test case metrics to better reflect the testing effort, include some well-known OO metrics in our study, explore the use of the Qi metric during the testing process in order to better guide the distribution of the testing effort, and finally replicate the study on various OO software systems to be able to give generalized results.

Acknowledgments

The authors would like to acknowledge the support of this paper by NSERC (National Sciences and Engineering Research Council of Canada) Grant. The authors would also like to thank the editor and anonymous reviewers for their very helpful comments and suggestions.

References

[1] P. L. Yeh and J. C. Lin, "Software testability measurement derived from data flow analysis," in *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, Florence, Italy, 1998.

[2] B. Baudry, B. Le Traon, and G. Sunyé, "Testability analysis of a UML class diagram," in *Proceedings of the 9th International Software Metrics Symposium (METRICS '03)*, IEEE CS, 2003.

[3] B. Baudry, Y. Le Traon, G. Sunyé, and J. M. Jézéquel, "Measuring and improving design patterns testability," in *Proceedings of the 9th International Software Metrics Symposium (METRICS '03)*, IEEE Computer Society, 2003.

[4] M. Bruntink and A. van Deursen, "An empirical study into class testability," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1219–1232, 2006.

[5] L. Zhao, "A new approach for software testability analysis," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 985–988, May 2006.

[6] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society Press, 1990.

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

[8] M. Bruntink and A. Van Deursen, "Predicting class testability using object-oriented metrics," in *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '04)*, pp. 136–145, September 2004.

[9] V. Gupta, K. K. Aggarwal, and Y. Singh, "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability," *Journal of Computer Science*, vol. 1, no. 2, pp. 276–282, 2005.

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

[11] Y. Singh, A. Kaur, and R. Malhota, "Predicting testability effort using artificial neural network," in *Proceedings of the World Congress on Engineering and Computer Science*, San Francisco, Calif, USA, 2008.

[12] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software Quality Journal*, vol. 18, no. 1, pp. 3–35, 2009.

[13] L. Badri, M. Badri, and F. Touré, "Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems," in *Advances in Software Engineering*, T.-h. Kim, H.-K. Kim, M. K. Khan et al., Eds., vol. 117 of *Communications in Computer and Information Science*, Springer, Berlin, Germany, 2010.

[14] L. Badri, M. Badri, and F. Touré, "An empirical analysis of lack of cohesion metrics for predicting testability of classes," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 2, 2011.

[15] M. Badri and F. Touré, "Empirical analysis for investigating the effect of control flow dependencies on testability of classes," in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE '11)*, 2011.

[16] M. Badri, L. Badri, and F. Touré, "Empirical analysis of object-oriented design metrics: towards a new metric using control flow paths and probabilities," *Journal of Object Technology*, vol. 8, no. 6, pp. 123–142, 2009.

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

[18] J. Gao and M. C. Shih, "A component testability model for verification and measurement," in *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC '05)*, pp. 211–218, July 2005.

[19] J. W. Sheppard and M. Kaufman, "Formal specification of testability metrics in IEEE P1522," in *Proceedings of the IEEE Systems Readiness Technology Conference Autotestcom (AUTOTESTCON '01)*, pp. 71–82, Valley Forge, Pa, USA, August 2001.

[20] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[21] J. M. Voas, "PIE: a dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, 1992.

[22] J. M. Voas and K. W. Miller, "Semantic metrics for software testability," *The Journal of Systems and Software*, vol. 20, no. 3, pp. 207–216, 1993.

[23] J. M. Voas and K. W. Miller, "Software testability: the new verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, 1995.

[24] R. V. Binder, "Design for testability in object-oriented systems," *Communications of the ACM*, vol. 37, no. 9, 1994.

[25] T. M. Khoshgoftaar, R. M. Szabo, and J. M. Voas, "Detecting program modules with low testability," in *Proceedings of the 11th IEEE International Conference on Software Maintenance*, pp. 242–250, October 1995.

[26] T. M. Khoshgoftaar, E. B. Allen, and Z. Xu, "Predicting testability of program modules using a neural network," in *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and SE Technology*, 2000.

[27] J. McGregor and S. Srinivas, "A measure of testing effort," in *Proceedings of the Conference on Object-Oriented Technologies*, pp. 129–142, USENIX Association, June1996.

[28] A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 97–108, 1996.

[29] Y. Le Traon and C. Robach, "Testability analysis of co-designed systems," in *Proceedings of the 4th Asian Test Symposium (ATS '95)*, IEEE Computer Society, Washington, DC, USA, 1995.

[30] Y. Le Traon and C. Robach, "Testability measurements for data flow designs," in *Proceedings of the 4th International Software Metrics Symposium*, pp. 91–98, Albuquerque, NM, USA, November 1997.

[31] Y. Le Traon, F. Ouabdesselam, and C. Robach, "Analyzing testability on data flow designs," in *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE '00)*, pp. 162–173, October 2000.

[32] A. Petrenko, R. Dssouli, and H. Koenig, "On evaluation of testability of protocol structures," in *Proceedings of the International Workshop on Protocol Test Systems (IFIP '93)*, Pau, France, 1993.

[33] K. Karoui and R. Dssouli, "Specification transformations and design for testability," in *Proceedings of the IEEE Global Elecommunications Conference (GLOBECOM '96)*, 1996.

[34] S. Jungmayr, "Testability measurement and software dependencies," in *Proceedings of the 12th International Workshop on Software Measurement*, October 2002.

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

[36] T. B. Nguyen, M. Delaunay, and C. Robach, "Testability analysis applied to embedded data-flow software," in *Proceedings of the 3rd International Conference on Quality Software (QSIC '03)*, 2003.

[37] B. Baudry, Y. Le Traon, and G. Sunyé, "Improving the testability of UML class diagrams," in *Proceedings of the International Workshop on Testability Analysis (IWoTA '04)*, Rennes, France, 2004.

[38] V. Chowdhary, "Practicing testability in the real world," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, IEEE Computer Society Press, 2009.

[39] R. A. Khan and K. Mustafa, "Metric based testability model for object-oriented design (MTMOOD)," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 2, 2009.

[40] A. Kout, F. Touré, and M. Badri, "An empirical analysis of a testability model for object-oriented programs," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 4, 2011.

[41] Y. Singh and A. Saha, "Predicting testability of eclipse: a case study," *Journal of Software Engineering*, vol. 4, no. 2, 2010.

[42] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[43] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[44] K. K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study," *Software Process Improvement and Practice*, vol. 14, no. 1, pp. 39–62, 2009.

[45] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: a multiple case study," in

Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09), pp. 291–301, October 2009.

[46] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, pp. 209–218, March 2009.

[47] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "SCOTCH: test-to-code traceability using slicing and conceptual coupling," in *Proceedings of the International Conference on Software Maintenance (ICSM '11)*, 2011.

[48] M. H. Halstead, *Elements of Software Science*, Elsevier/North-Holland, New York, NY, USA, 1977.

[49] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[50] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

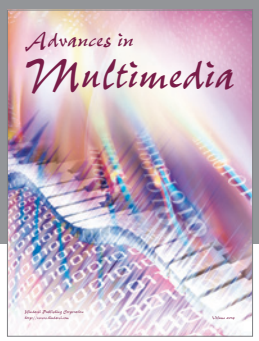
[51] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

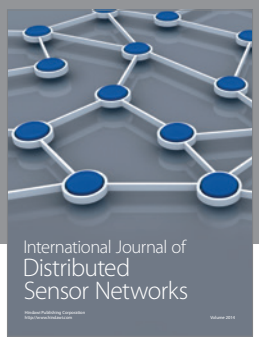
[52] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

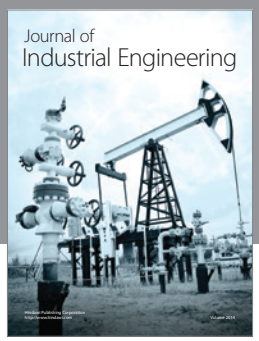
[53] K. El Emam and W. Melo, "The prediction of faulty classes using object-oriented design metrics," National Research Council of Canada NRC/ERB 1064, 1999.

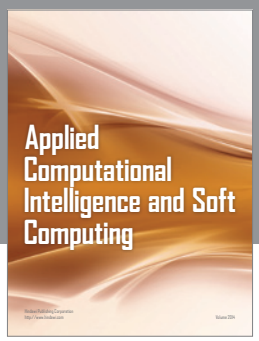
[54] D. Hosmer and S. Lemeshow, *Applied Logistic Regression*, Wiley-Interscience, 2nd edition, 2000.

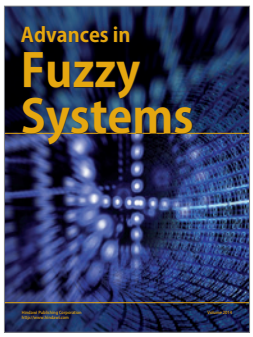
[55] K. El Emam, "A Methodology for validating software product metrics," National Research Council of Canada NRC/ERB 1076, 2000.

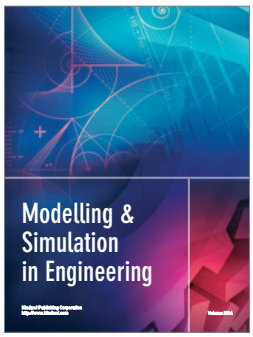


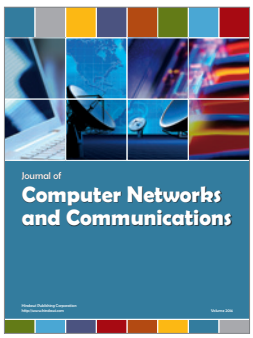












Hindawi
Submit your manuscripts at
http://www.hindawi.com