Analyzing Software Quality Evolution using Metrics: An Empirical Study on Open Source Software

Nicholas Drouin
Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Québec, Canada
Email: Nicholas.Drouin@uqtr.ca

Mourad Badri and Fadel Touré
Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Québec, Canada
Email: {Mourad.Badri, Fadel.Toure}@uqtr.ca

Abstract—**The study presented in this paper aims at analyzing empirically the quality of evolving software systems using metrics. We used a synthetic metric (Quality Assurance Indicator - Qi), which captures in an integrated way different object-oriented software attributes. We wanted to investigate if the Qi metric can be used to observe how quality evolves along the evolution of software systems. We consider software quality from an internal (structural) perspective. We used various object-oriented design metrics for measuring the structural quality of a release. We performed an empirical analysis using historical data collected from successive released versions of three open source (Java) software systems. The collected data cover, for each system, a period of several years (4 years for two systems and 7 years for the third one). We focused on three issues: (1) the evolution of the Qi metric along the evolution of the subject systems, (2) the class growth of the subject systems, and (3) the quality of added classes versus the quality of removed ones. Empirical results provide evidence that the Qi metric reflects properly the quality evolution of the studied software systems.**

Index Terms—**Software Evolution, Software Quality, Software Attributes, Object-Oriented, Metrics.**

I. Introduction

As software systems are used for a long period of time, software evolution is inevitable. Software evolution is, in fact, the dynamic behavior of programming systems as they are maintained and enhanced over their lifetimes [1]. Indeed, software systems need to continually evolve during their life cycle for various reasons: adding new features to satisfy user requirements, changing business needs, introducing novel technologies, correcting faults, improving quality, etc. [2, 3].

Software evolution plays a key role in the overall lifecycle of a software system. Furthermore, it takes a large part of the overall lifecycle costs. So, as software evolves, the changes made to the software must be carefully managed. The accumulation of changes through the successive versions of a software system can, in fact, lead to a degradation of its quality [4-8]. This can also lead to an increasing time of requirements implementation over a longer period of time. It is, therefore, important to keep software quality under control during evolution. Software quality is a complex notion that cannot be defined in a simple way. The definition of software quality includes various attributes such as functionality, usability and maintainability [9]. Quality plays an important role in a software project's success. So, as software evolves, it is important to monitor how its quality changes so that quality assurance (QA) activities can be properly planned [8].

Software metrics play an essential part in understanding and controlling the overall software engineering process [2]. Software metrics can, particularly, be used to analyze the evolution of the quality of software systems [10]. Metrics have, in fact, a number of interesting characteristics for providing evolution support [11]. A large number of metrics have been proposed for measuring various attributes of object-oriented (OO) software systems such as size, inheritance, complexity, cohesion and coupling [12]. Furthermore, many studies have been performed to investigate the relationships between OO metrics and software quality attributes. Software metrics can be, in fact, useful in assessing (predicting) software quality and supporting various software engineering activities [13, 14]. However, with the growing complexity and size of OO software systems, the ability to reason about such a major issue using synthetic metrics, compared to using several metrics simultaneously, would be more appropriate in practice.

We proposed in [15] a new metric, called Quality Assurance Indicator (Qi), capturing in an integrated way the interactions between classes and the distribution of the control flow in a software system. The Qi of a class is based on different intrinsic characteristics of the class, as well as on the Qi of its collaborating classes. It is important to notice, however, that this metric has no ambition to capture the overall quality of OO software systems. Furthermore, the objective is not to evaluate a design by giving absolute values, but more relative values that may be used for identifying critical classes on which more QA effort is needed to ensure software quality.

doi:10.4304/jsw.8.10.2462-2473

Applying equal QA effort to all classes of a software system is, indeed, cost-prohibitive and not realistic, particularly in the case of large and complex software systems. In [15], we performed an empirical analysis using data collected from several open source (Java) software systems. In all, more than 4,000 classes were analyzed (400 000 lines of code). We compared the Qi metric, using the Principal Components Analysis (PCA) method, to various well known OO metrics. The selected metrics were grouped in five categories: coupling, cohesion, inheritance, complexity and size. Empirical results provide evidence that the Qi metric captures, in a large part, the information provided by the studied OO metrics. Moreover, we explored in [16] the relationship between the Qi metric and testability of classes and investigated in [17] the capacity of the Qi metric in predicting the unit testing effort of classes using regression analysis. Results provide evidence that the Qi metric is able to accurately predict the unit testing effort of classes. More recently, we explored in [18] if the Qi metric can be used to observe how quality, measured in terms of defects, evolves in the presence of changes and in [19] if the Qi metric captures the evolution of two important OO metrics (related to coupling and complexity).

The study presented in this paper aims at investigating more deeply if the Qi metric can be used to observe how quality evolves along the evolution of a software system. We focus on retrospective analysis of software quality. We consider software quality from an internal (structural) perspective. We explored, in particular, three issues related to software evolution: (1) Evolution of the Qi metric: we wanted to investigate if the Qi metric can be used to observe the evolution of software quality from an internal perspective. (2) Class growth: we wanted to explore the relationship between the Qi metric and the size growth along the different released versions of the subject systems. (3) Quality of added versus removed classes: we wanted to investigate how the Qi metric captures the quality of these classes, and compare the quality of added classes to the quality of removed ones. In order to capture the internal quality of a release, we used various well-known OO design metrics, specifically the Chidamber and Kemerer (CK) metrics suite [20]. We also include in our study the size related LOC (Lines Of Code) metric. Empirical evidence exists, in fact, showing that there exist a relationship between these metrics and different software quality attributes [10, 13, 16, 21-27]. We performed an empirical analysis using historical data collected from successive released versions of three open source (Java) software systems. The collected data cover, for each system, a period of several years. Empirical results provide evidence that the Qi metric reflects properly the quality evolution of the subject software systems.

The rest of this paper is organized as follows: Section 2 gives a survey on related work. The Qi metric is introduced in Section 3. Section 4 presents the OO design metrics we used in our study in order to capture the internal quality of a release. Section 5 presents the empirical study we performed to evaluate the ability of the Qi metric to reflect (capture) the evolution of software quality. Finally, Section 6 summarizes the contributions of this work and outlines directions for future work.

II. Related works

Mens et *al*. [11] provide an overview of the ways metrics have been, and can be, used to analyze software evolution. The authors argued that software metrics have a number of interesting characteristics for providing evolution support. Mens et *al*. make a distinction between the use of software metrics before the evolution has occurred (*i.e.*, predictive) and after the evolution has occurred (*i.e.*, retrospective). To support retrospective analysis, metrics can be used to understand the quality evolution of a software system by considering its successive releases. In particular, metrics can be used to measure whether the quality of a software has improved or degraded between two releases.

Dagpinar et *al*. [22] investigate the significance of various OO metrics for the purpose of predicting software maintainability. The used metrics have been categorized within four groups: size, inheritance, cohesion and coupling metrics. Dagpinar et *al*. used historical data on the maintenance history of two software systems over a period of three years. Nagappan et *al*. [28] focus on mining metrics to predict component failures. In an empirical study of the post-release defect history of five Microsoft software systems, the authors found that failure-prone software entities are statistically correlated with code complexity measures. Nagappan et *al*. noted, however, that there is no single set of complexity metrics that could be used as a universally best defect predictor.

Ambu et *al*. [29] focus on the evolution of quality metrics in an agile/distributed project. The project has been monitored on a regular basis collecting both process and product metrics. The product metrics include the CK [20] suite of quality metrics. By analyzing the evolution of these metrics, the authors investigated how the distribution of the development team has impacted the source code quality. Lee et *al*. [10] provide an overview of open source software evolution with software metrics. The authors argue that software metrics can be used to assess the quality along the evolution of a software system. Lee et *al*. explored the evolution of an open source software system in terms of size, coupling and cohesion, and discussed its quality change based on the Lehman's laws of evolution [4, 5, 30]. Software metrics were derived from various releases of the open source software system studied.

Jermakovics et *al*. [31] propose an approach to visually identify software evolution patterns related to requirements. A combined visualization showing the evolution of a software system with the implementation of its requirements is proposed. The authors argue that such view can help project managers to keep the evolution process of a software system under control. Jermakovics et *al*. used in their work complexity, coupling and cohesion metrics as defined by Chidamber et *al*. [20]. Mens et *al*. [32] present a metrics-based study

about the evolution of Eclipse. The authors consider seven major releases and investigate whether three of the Lehman's laws of software evolution were supported by the data collected. Mens et *al*. focused on continuing change, increasing complexity and continuing growth.

Xie et *al*. [3] conduct an empirical analysis on the evolution of seven open source programs. The study investigates (and validates some of) Lehman's laws of evolution. The authors found that different branches of open source programs evolve in parallel. They also found similarities in the evolution patterns of the programs studied. Xie et *al*. used source code metrics as well as project and defect information to analyze software growth, characterize software changes, and assess software quality.

Murgia et *al*. [24] focus on software quality evolution in open source projects using agile practices. The authors used a set of OO metrics to study software evolution and its relationship with bug distribution. According to the achieved results, they conclude that there is no a single metric that is able to explain the bug distribution during the evolution of the analyzed systems. Zhang et *al*. [8] use c-charts and patterns to monitor quality evolution over a long period of time. The number of defects was used as a quality indicator. Two open source software systems (Eclipse (Java) and Gnome (C++)) have been used to illustrate the approach. Zhang et *al*. identified six common quality evolution patterns (downward trend, upward trend, impulse, hills, small variations and roller coaster), and argue that quality evolution patterns are useful for prioritizing QA efforts in practice.

Recently, Eski et *al*. [23] present an empirical study on the relationship between OO metrics and changes in software. The authors analyze modifications in software across the historical sequence of open source projects and propose a metric-based approach to predict change-prone classes. The authors used QMOOD [33] and CK [20] metrics. Yu et *al*. [34] study the possibility of using the number of bug reports as a software quality measure. Using statistical methods, the authors analyze the correlation between the number of bug reports and software changes.

III. Quality Assurance Indicator

In this section, we give a summary of the definition of the Quality Assurance Indicator (Qi) metric. For more details see [15, 17, 18]. The Qi metric is based on the concept of Control Call Graphs (CCG), which are a reduced form of traditional Control Flow Graphs (CFG). A CCG is, in fact, a CFG from which the nodes representing instructions (or basic blocks 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 QA 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).

A. Control Call Graphs

Let us consider the example of method M given in Fig. 1.1. The S_i represents blocks of instructions that do not contain a call to a method. The code of method M reduced to control call flow is given in Fig. 1.2. The instructions (blocks of instructions) not containing a call to a method are removed from the original code of method M. Fig. 1.3 gives the corresponding CCG. Compared to traditional Call Graphs, CCGs are much more precise models. CCGs capture, in fact, the structure of calls and related control.

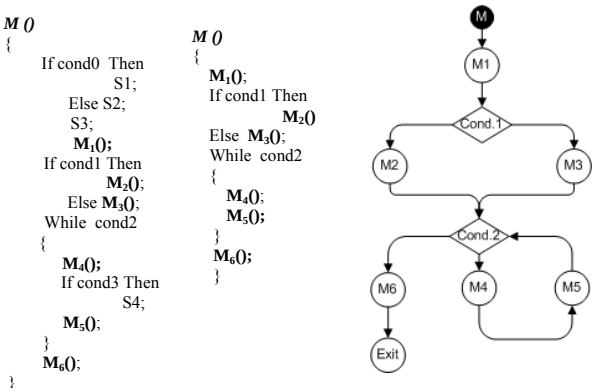

Figure 1. A method and its corresponding Control Call Graph.

B. Quality Assurance Indicator

The Qi of a method M_i is defined 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 on intrinsic characteristics of 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 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 (which is not captured by traditional OO metrics). The Qi of a method M_i is given by:

$$\mathrm{Qi}_{\mathrm{M_i}} = \mathrm{Qi}^{*}_{\mathrm{M_i}} \cdot \sum_{\mathrm{j}=1}^{\mathrm{n_i}} \left[\mathrm{P}\left(\mathrm{C}_{\mathrm{j}}^{\mathrm{i}}\right) \cdot \prod_{\mathrm{M} \in \sigma_{\mathrm{j}}} \mathrm{Qi}_{\mathrm{M}} \right] \qquad (1)$$

with :

$\mathrm{Qi}_{\mathrm{M_i}}$: QA indicator of method M_i,
$\mathrm{Qi}^{*}_{\mathrm{M_i}}$: intrinsic QA indicator of method M_i,
C_j^i: j^{th} path of method M_i,
$P(C_j^i)$: probability of execution of path C_j^i of method M_i,
Qi_{M}: QA indicator of the method M included in the path C_j^i,
n_i: number of linear paths of the CCG 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 (number of methods in the program) equations. 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 C (noted Qi_C) as the product of the Qi of its methods:

$$Qi_C = \prod_{M \in \delta} Qi_M \quad (2)$$

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

C. Assigning Probabilities

The CCG 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:

$$P(C) = \prod_{A \in \theta} P(A) \quad (3)$$

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.

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	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.

To facilitate our experiments, 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. These values can be adapted according to the nature of the applications (for example). As an alternative way, the probability values may also be assigned (adapted) by programmers during the development (in an iterative way, knowing the code) or obtained by dynamic analysis. Dynamic analysis is out of the scope of this paper.

D. Intrinsic Quality Assurance Indicator

The *Intrinsic Quality Assurance Indicator* of a method M_i, noted Qi^*_{Mi}, is given by:

$$Qi^*_{M_i} = (1 - F_i) \quad (4)$$

with: $F_i = \frac{(1-tc_i)cc_i}{cc_{max}}$

where:

CC_i : cyclomatic complexity of method M_i,

$cc_{max} = \max_{1 \le i \le N}(cc_i)$

tc_i: unit testing coverage of the method M_i, $tc_i \in [0,1]$.

Many studies provided empirical evidence that there is a significant (and strong) relationship between cyclomatic complexity and fault proneness (e.g., [13, 27, 35]). Testing (as one of the most important QA) 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. The testing coverage measures are, currently in our approach, affected by programmers based on the test suites developed for the classes of the system. The testing coverage measures can also be obtained automatically (using tools such as Together (www.borland.com) or CodePro (developers.google.com)) by analyzing the code of the test suites (JUnit suites for example) to determine which parts of the classes that are covered by the test suites and those that are not. This issue is out of the scope of this paper and will be considered in our future work.

IV. OBJECT-ORIENTED DESIGN METRICS

We present, in this section, the summary of the OO design metrics we used in our empirical study in order to measure the structural quality of a release. These metrics have been selected for study because they have received considerable attention from researchers, are being increasingly adopted by practitioners, and are incorporated into several development tools. Furthermore, empirical evidence exists on the relationship between these metrics and various software quality attributes. In addition, some of these metrics have been used in several studies on software evolution. We selected in total seven metrics. Six of these metrics (CBO, LCOM, DIT, NOC, WMC and RFC) were proposed by Chidamber and Kemerer [20, 36]. We also include the size related LOC metric. We give in what follows a brief definition of each metric.

Coupling Between Objects: The CBO metric counts for a class the number of other classes to which it is coupled (and vice versa).

Lack of Cohesion in Methods: The LCOM metric measures the dissimilarity of methods in a class. It is defined as follows: LCOM = |P| - |Q|, if |P| > |Q|, where P is the number of pairs of methods that do not share a common attribute and Q the number of pairs of methods sharing a common attribute. If the difference is negative, LCOM is set to 0.

Depth of Inheritance Tree: The DIT metric of a class is given by the length of the (longest) inheritance path from the root of the inheritance hierarchy to the class on which it is measured (number of ancestor classes).

Number Of Children: The NOC metric simply measures the number of immediate subclasses of the class in a hierarchy.

Response For Class: The RFC metric for a class is defined as the set of methods that can be executed in response to a message received by an object of the class.

Weighted Methods per Class: The WMC metric gives the sum of complexities of the methods of a given class, where each method is weighted by its cyclomatic complexity. Only methods specified in the class are considered.

Lines Of Code per class: The traditional LOC metric, which has been used for a large number of software development activities, is widely accepted as a size/complexity metric. It counts for a class its number of lines of code.

V. Experimental Study

We present, in this section, the empirical study we conducted in order to investigate if the Qi metric can be used to observe how quality evolves along the evolution of a software system. In particular, we wanted to investigate if the Qi metric reflects properly the improvement (or degradation) of the quality, from an internal perspective, of a software between two releases and along its evolution. We used historical data collected from successive released versions of three open source (Java) software systems. We focused on retrospective analysis of software quality. We explored, in particular, the three following issues: (1) the evolution of the Qi metric along the evolution of the subject systems, (2) the class growth of the subject systems, and (3) the quality of the added classes versus the quality of removed ones.

A. Empirical Design

(1) Selection of the subject systems

We selected our subject systems based on the following requirements:

- Source code archives of the subject systems must be important enough to provide a significant data set on the evolution of the systems.
- Subject systems must be of different overall size and quality, in order to see if our results will differ from one system to another.
- Subject systems must be developed in Java (the tool we developed is for Java applications).

We selected for our study two Eclipse components (JDT.Debug and PDE.UI). These two components have been used in an empirical study conducted by Zhang et *al*. [8] in order to monitor software quality evolution over a long period of time. Zhang et *al*. addressed software quality from an external point of view and used the number of defects as a quality indicator. Zhang et *al*. showed that these two systems follow distinct evolution patterns. The authors identified, in fact, six common quality evolution patterns (downward trend, upward trend, impulse, hills, small variations and roller coaster). Zhang et *al*. also showed that JDT.Debug follows an upward quality pattern while PDE.UI follows a roller coaster quality pattern. These captures are available throughout CVS. Captures were taken at regular intervals (monthly) during a wide lapse of time (more than four years).

TABLE II.
SOME STATISTICS ON THE USED SYSTEMS.

Systems		**Eclipse JDT.Debug**	**Eclipse PDE.UI**	**Apache Tomcat**
Time frame (years)		4.25	4.25	7.2
Releases /Captures		52 (monthly)	52 (monthly)	31 (official)
First release computed	**Version**	-	-	5.5.0
	Date	2002-08-29	2002-08-29	-
	Size (SLOC)	12 201	9 519	126 927
Last release computed	**Version**	-	-	5.5.35
	Date	2006-06-26	2006-11-28	-
	Size (SLOC)	31 884	79 548	170 998

TABLE III.
VALUES OF THE SELECTED METRICS FOR JDT.DEBUG.

Metrics	**First version**	**Last version**
Number of classes	127	218
Total KLOC	12.2	31.9
Avg. Qi	0.791	0.796
Avg. CBO	10.3	14.1
Avg. DIT	2.64	2.83
Avg. RFC	80.2	88.9
Avg. NOC	2.70	1.95
Avg. LCOM	184	358
Avg. WMC	20.0	28.7
Avg. LOC	96.1	146

TABLE IV.
VALUES OF THE SELECTED METRICS METRICS FOR PDE.UI.

Metrics	**First version**	**Last version**
Number of classes	121	670
Total KLOC	9.52	79.5
Avg. Qi	0.774	0.725
Avg. CBO	2.61	23.2
Avg. DIT	0.876	3.02
Avg. RFC	15.5	93.4
Avg. NOC	1.00	1.07
Avg. LCOM	9.71	48.6
Avg. WMC	8.48	22.4
Avg. LOC	78.7	119

TABLE V.
VALUES OF THE SELECTED METRICS FOR TOMCAT.

Metrics	**First version**	**Last version**
Number of classes	837	1108
Total KLOC	126	171
Avg. Qi	0.743	0.735
Avg. CBO	8.97	9.41
Avg. DIT	1.60	1.64
Avg. RFC	60.5	65.9
Avg. NOC	0.686	0.661
Avg. LCOM	179	184
Avg. WMC	29.8	30.3
Avg. LOC	152	154

We also selected for our study a third (relatively large) system, Tomcat, which is an open source web server developed by the Apache Software Foundation. We analyzed its 5.5 branch, launched in August 2004. The version 5.5.35, the latest to date, was launched on November 2011. For this system, we used the official releases as time captures. Table II gives some statistics on the three subject systems. Tables III, IV and V give the average values of the selected metrics for the first and last versions of each subject system (respectively JDT.Debug, PDE.UI and Tomcat).

(2) Data Gathering

We collected three types of data from the subject systems: source code historical data, OO metrics data and Qi data.

System history data: We used CVS (Concurrent Versions System) to collect historical data about two of the subject systems. CVS is a client-server software that allows keeping track of the evolution of a software system. For PDE.UI and JDT.Debug, we connected to the CVS repertory of Eclipse. We based our analysis on the latest available version on the first day of each month. A period of more than four years (fifty-two versions in total) is covered for both systems. For Apache Tomcat, we retrieved the official releases on the 5.5.x branch from the official website (archive.apache.org/dist/tomcat/tomcat-5). A period of more than seven years (thirty-one versions in total) is covered for this system.

Metrics data: We used the Borland Together tool (www.borland.com/) to collect data on the OO metrics. For each released version of a subject system, we computed the metrics values at the micro level (classes) as well as at the macro level (system). We used the average as an aggregation method.

Qi data: We used the tool we developed to collect the Qi data. We computed the Qi value for each class of each released version of a subject system. Here also, we computed the Qi values at the two levels (micro and macro). These data have been merged with the OO metrics data to group the data set for each class (and released version of a subject system). For our experiments, since we did not have any data on the test suites used for testing the subject systems and knowing that the main purpose of this study is to explore if the Qi metric can be used to observe how quality evolves along the evolution of the subject systems, the testing coverage (tci, Section 3.2) is set to 0.75 for all methods. As mentioned previously, the objective of the Qi metric is not to evaluate a design by giving absolute values, but more relative values that may be used for identifying, in a relative way, the critical classes on which more QA effort is required to ensure software quality (iterative distribution of the QA effort).

B. Internal Software Quality Evolution

In this section, we investigate the evolution of the Qi metric along the evolution of the subject systems. We wanted to explore if the Qi metric can be used to observe the evolution of software quality from an internal perspective. In order to capture the internal quality of a release, we used the OO design metrics.

(1) Hypothesis

We tested the following hypothesis:

Hypothesis 1: The evolution of the structural quality, measured using OO metrics, along the evolution of the released versions of a subject system will be reflected positively by the Qi metric on both micro and macro levels.

(2) Metrics Evolution

We applied the selected metrics to the classes of the three software systems of our study. We calculated the average values of the metrics for each released version from each system. The analysis of the collected data allowed us to observe the overall trend of the Qi and OO metrics along the evolution of each system. Figs. 2, 3 and 4 show the evolution of the Qi and OO metrics (shown with the min-max normalization) for the three systems under study.

JDT.Debug (Fig. 2) shows significant variations (increase and decrease) along its evolution. The two curves (Qi and OO metrics) have an evolution characterized by different levels (steps). We can actually observe a period of several iterations where the variations are minimal. OO metrics generally seem to follow an upward curve. The Qi values show significant increases neutralized by subsequent reductions few iterations later.

PDE.UI (Fig. 3) shows uniformity in the curves of the OO metrics. This trend is strongly positive, especially because of the significant increase occurring at iteration 16. However, despite this significant increase, a growth can still be observed thereafter. Consequently, Qi follows the opposite trend. At iteration 16, we can observe a significant decrease for the Qi, after which a tray is then present for several iterations.

Apache Tomcat (Fig. 4) also shows a similar trend for all the metrics which is characterized by a steady increase and by some fluctuations (peaks). Qi curve shows a slight growth in the first iterations, which becomes however a decrease around iteration 8. This negative growth appears in a relatively continuous way.

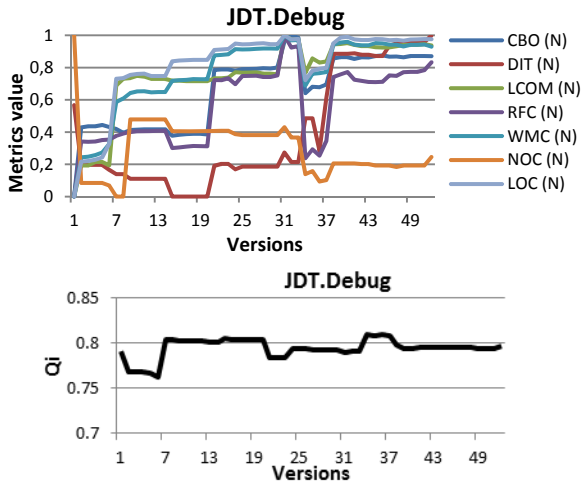

Figure 2. Evolution of the Qi and OO metrics for JDT.Debug.

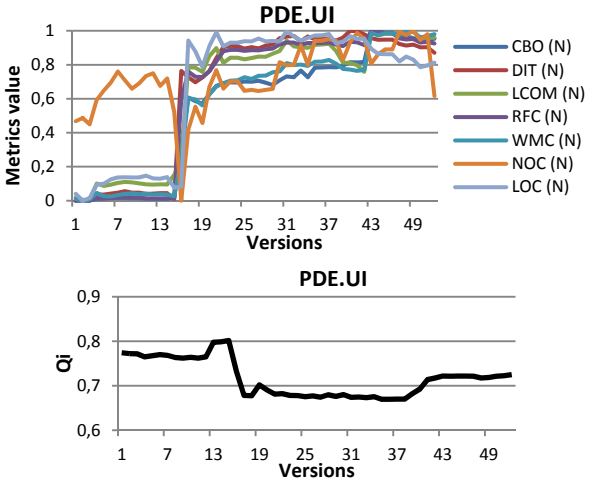

Figure 3. Evolution of the Qi and OO metrics for PDE.UI.

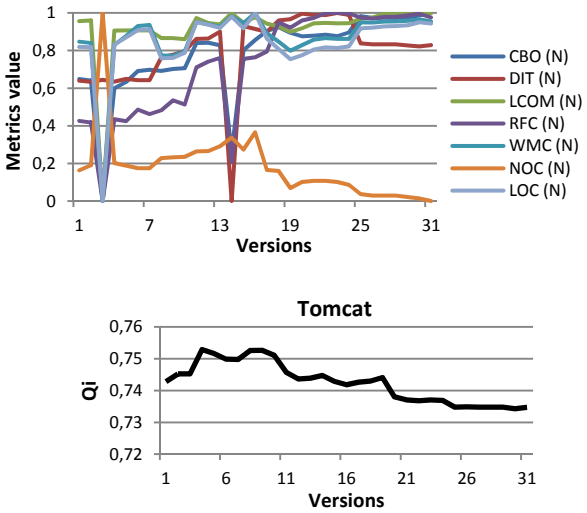

Figure 4. Evolution of the Qi and OO metrics for Tomcat.

(3) Correlations Analysis

In order to validate Hypothesis 1, we analyzed the correlations between the Qi metric and the selected OO metrics along the evolution of the released versions of the subject systems. We performed statistical tests using correlation. We used both Spearman's and Pearson's correlation coefficients in our study. These techniques are widely used for measuring the degree of relationship between two variables. Correlation coefficients will take a value between -1 and +1. A positive correlation is one in which the variables increase (or decrease) together. A negative correlation is one in which one variable increases as the other variable decreases. A correlation of +1 or -1 will arise if the relationship between the variables is exactly linear. A correlation close to zero means that there is no linear relationship between the variables. We used the XLSTAT (www.xlstat.com/) tool to perform the analysis. We applied the typical significance threshold ($\alpha = 0.05$) to decide whether the correlations were significant. We analyzed the correlations between the Qi metric and the selected OO metrics at both levels: micro and macro.

For the micro level, we selected from each studied system the classes present from the first version of the system to its latest one. We considered that these classes represent in some ways the core of the system throughout the period of evolution analyzed. We give in what follows (for space reasons) only the results obtained in the case of Tomcat, because of its size much larger than the size of the other two systems. As mentioned previously, a period of more than seven years (thirty-one versions in total) is covered for Tomcat. Fig. 5 shows the evolution of the correlation values (Pearson) between the Qi and OO metrics (at the micro level) along the different versions of Tomcat. Table VI shows the average value of these correlations (the significant correlations are in boldface).

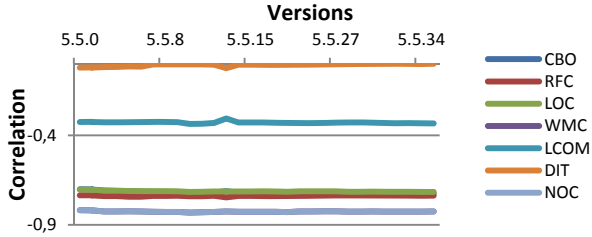

Figure 5. Evolution of correlations values at the micro level between Qi and OO metrics for Tomcat.

TABLE VI.
CORRELATIONS MEAN VALUES AT THE MICRO LEVEL BETWEEN QI AND OO METRICS FOR TOMCAT.

OO Metrics	Correlation
CBO	**-0.7178**
RFC	**-0.7384**
LOC	**-0.7129**
WMC	**-0.8260**
LCOM	**-0,3288**
DIT	-0,0075
NOC	**-0,8260**

It can be seen, from Fig. 5 and Table VI, that the obtained correlations values at the micro level between the Qi and OO metrics are relatively high, and this along the evolution of Tomcat (except for LCOM (the correlation is significant but relatively low) and DIT (the correlation is not significant)). Moreover, the correlations values remain relatively stable from one iteration to another (Fig. 5), aside the slight peak observed for version 5.5.13. Several observations on the computed metrics can explain this fluctuation. In comparison with version 5.5.12, the size in terms of lines of code of version 5.5.13 increases (from 148 900 to 149 500), while the number of classes decreases (from 968 to 964), which has the effect of increasing the average LOC per class (from 153.8 to 155.1), decreasing the coupling (CBO from 9.19 to 8.49) and increasing the number of operations per class (NOO from 12.9 to 13.1). We can therefore explain these observations by an amount of added instructions which is concentrated in a (relative) small number of classes.

Moreover, the correlations values between the Qi and OO metrics are negative. A negative correlation indicates that one variable (Qi metric in our case) decreases as the other variable (OO metric in our case) increases. These results are plausible and not surprising. In fact, the more strongly a class is coupled to other classes (and its complexity and size are high), the less the quality of the class is likely to be. A low value of the Qi of a class (high value of coupling and complexity) indicates as mentioned previously that the class is a high-risk class and needs a high QA effort to ensure its quality. These results and observations suggest that the Qi metric, at the micro level, captures the evolution of the selected OO metrics. These results support therefore Hypothesis 1 at the micro level.

For the macro level, we used the average values of the metrics for each version. Tables VII, VIII and IX show the correlations (Spearman) values obtained between the Qi and OO metrics for the three systems studied. The correlations that are significant are in boldface. Here also, we applied the typical significance threshold ($\alpha = 0.05$) to decide whether the correlations were significant.

TABLE VII.
CORRELATIONS VALUES BETWEEN THE QI AND OO METRICS AT THE MACRO LEVEL FOR JDT.DEBUG.

Variables	Qi
CBO	**-0.373**
DIT	-0.233
RFC	**-0.443**
NOC	0.062
LCOM	-0.007
WMC	-0.127
LOC	-0.138

TABLE VIII.
CORRELATIONS VALUES BETWEEN THE QI AND OO METRICS AT THE MACRO LEVEL FOR PDE.UI.

Variables	Qi
CBO	**-0.475**
DIT	**-0.681**
RFC	**-0.516**
NOC	**-0.285**
LCOM	**-0.554**
WMC	**-0.539**
LOC	**-0.891**

TABLE IX.
CORRELATIONS VALUES BETWEEN THE QI AND OO METRICS AT THE MACRO LEVEL FOR TOMCAT.

Variables	Qi
CBO	**-0.854**
DIT	**-0.464**
RFC	**-0.830**
NOC	**0.736**
LCOM	**-0.747**
WMC	**-0.566**
LOC	**-0.397**

From Tables VII, VIII and IX, it can be seen that the correlations between Qi and OO metrics are significant (in boldface) with all the metrics for PDE.UI and Tomcat. This suggests that the Qi metric captures the evolution of the OO metrics for an evolving system at the macro level. However, the results obtained in the case of system JDT.Debug are less conclusive. Only few significant correlations emerge for this system (between Qi and CBO and RFC). This may be due to the size of the system which is less important than the size of the other two systems. Moreover, the fact that the values used in this case are obtained by aggregation (mean values of classes) may also bias the results. Such observations thus allow us reasonably to validate Hypothesis 1 at the macro level.

C. Class Growth

In this section, we analyze the class growth of our three subject systems. We wanted to investigate, in particular, if the size growth (measured using various metrics) along the different released versions of the subject systems will be captured positively by the Qi metric. We used several size indicators. We used the *number of lines of code in the system* (SLOC) and the *total number of classes* (SNOC) at the system level, and the *number of lines of code in a class* (LOC) and the *number of operation per class* (NOO) at the class level.

(1) Hypothesis

We tested in this section the following hypothesis:

Hypothesis 2: The class growth along the evolution of the released versions of a subject system will be captured positively by the Qi metric.

(2) Data Analysis

To investigate the relationship between the Qi metric and size indicators, we used the Spearman correlation under the same conditions as the previous step. Tables X, XI and XII show the values of the size indicators for the first and last versions of the three subject systems. Figs. 6, 7 and 8 show the evolution of the size indicators (SLOC, SNOC, LOC and NOO) respectively for JDT.Debug, PDE.UI and Tomcat. Tables XIII, XIV and XV show the correlations obtained between the Qi metric and the size indicators for all the captures of the studied systems.

TABLE X.
SIZE METRICS FOR THE FIRST AND LAST VERSIONS OF JDT.DEBUG.

	Version	SLOC	SNOC	LOC	NOO
First version	2002-04	12200	127	96.1	10.2
Last version	2006-07	31900	218	146	13.3

TABLE XI.
SIZE METRICS FOR THE FIRST AND LAST VERSIONS OF PDE.UI.

	Version	SLOC	SNOC	LOC	NOO
First version	2002-09	9 520	121	78.7	4.17
Last version	2006-12	79 500	670	119	8.86

TABLE XII.
SIZE METRICS FOR THE FIRST AND LAST VERSIONS OF TOMCAT.

	Version	SLOC	SNOC	LOC	NOO
First version	5.5.0	126 000	837	152	12.7
Last version	5.5.35	171 000	1108	154	13.0

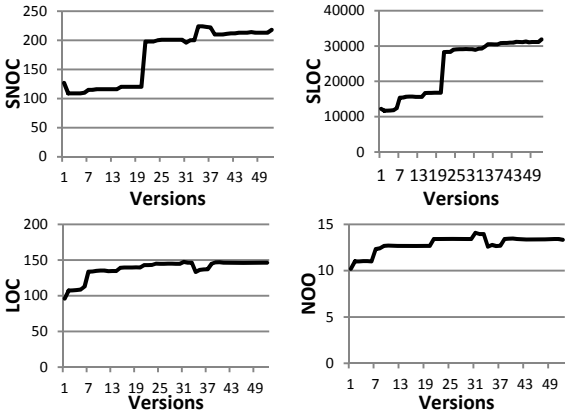

Figure 6. Evolution of size related metrics for JDT.Debug.

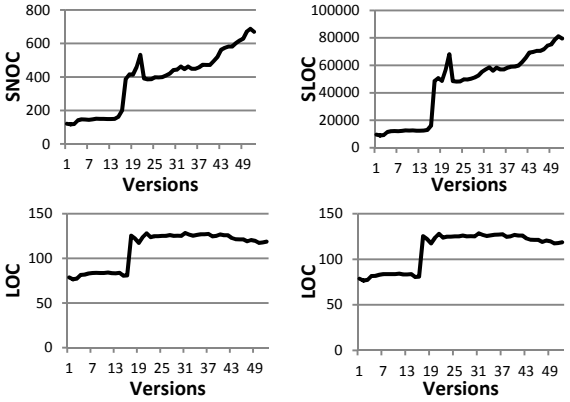

Figure 7. Evolution of size related metrics for PDE.UI.

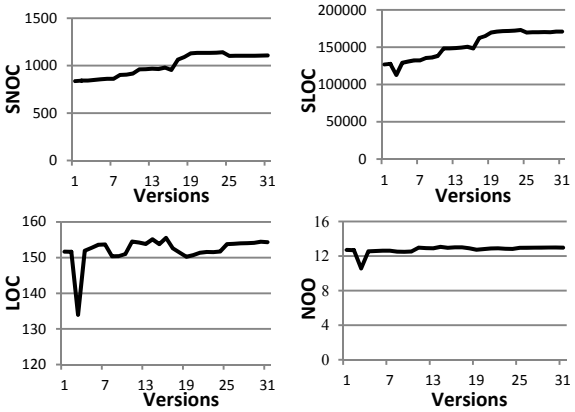

Figure 8. Evolution of size related metrics for Tomcat.

From Table X, it can be seen that for JDT.Debug, there is an increase in the number of classes by 72% between the first version analyzed (127) and the last one (218). The increase in PDE.UI (Table XI) is 454% (121 to 670) and 32% (837 to 1108) for Tomcat (Table XII). Figs. 6, 7 and 8 clearly reflect the size growth for the three systems. Size is shown (in these figures) using the four size indicators: SLOC, SNOC, LOC and NOO. It can be seen from Figs. 6, 7 and 8 that, for example, the number of classes (SNOC) clearly shows (for the three subject systems) an increasing evolution. This growth is fairly linear aside a few iterations where the increase is sudden (which could be explained by adding/removing large sections of code). The same trend is observed for the other size indicators (SLOC, LOC and NOO) for the three subject systems. To test Hypothesis 2, we calculated the correlations values between the average value of the Qi metric and the considered size indicators (SLOC, SNOC, LOC and NOO). Tables XIII, XIV and XV show the results. It can be seen that the correlations values are, overall, significant and relatively high for some of them. Correlations between the Qi metric and size metrics are all significant for PDE.UI and Tomcat. For JDT.Debug, the only significant correlation is between Qi and NOO. This may be due to the fact that the values used are obtained by aggregation (mean values of classes). Such observations thus allow us reasonably to validate Hypothesis 2.

TABLE XIII.
CORRELATIONS VALUES BETWEEN QI AND SIZE METRICS FOR JDT.DEBUG.

Size Indicator	Qi
SLOC	*0.129*
# Cls.	*0.219*
Avg. LOC	*-0.138*
Avg. NOO	**-0.321**

TABLE XIV.
CORRELATIONS VALUES BETWEEN QI AND SIZE METRICS FOR PDE.UI.

Size Indicator	Qi
SLOC	**-0.442**
# Cls.	**-0.433**
Avg. LOC	**-0.891**
Avg. NOO	**-0.561**

TABLE XV.
CORRELATIONS VALUES BETWEEN QI AND SIZE METRICS FOR TOMCAT.

Size Indicator	Qi
SLOC	**-0.790**
# Cls.	**-0.736**
Avg. LOC	**-0.397**
Avg. NOO	**-0.697**

D. Removed and Added Classes

In this section, we focused on the added/removed classes along the evolution of the subject systems. We wanted to explore how the Qi metric captures the quality of these classes, and compare the quality of added classes to the quality of removed ones. Here also, we used the OO design metrics to capture the structural quality of classes.

(1) Hypothesis

We tested in this section the following hypothesis:

Hypothesis 3: The difference in terms of quality (improvement versus degradation) between the added and removed classes of the subject systems will be reflected positively by the Qi metric.

(2) Data Analysis

For each version of a subject system, we identified the removed classes (group R) and the added ones (group A).

To achieve this, we compared each version to its previous and successive versions. Then, we associated to each version the values (selected source code metrics) of classes included in the two groups (with the average aggregation). Finally, the values obtained for the two groups (A and R), for each of the versions, are compared. Here also, and for the same reasons as those given in Section 5.2.3, we give in what follows only the results obtained in the case of Tomcat. Fig. 9 shows the number of added / removed classes for each version of Tomcat, starting with version 5.5.1.

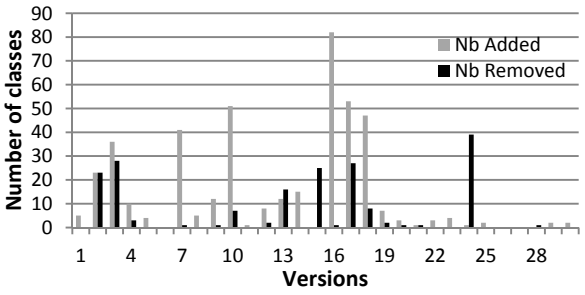

Figure 9. Number of added vs removed classes for each version of Tomcat.

Of all versions of Tomcat analyzed (30 in total), we included in this step only the versions for which there is at the same time added and removed classes. The other versions were excluded from this study. Table XVI provides details on these versions apart.

TABLE XVI.
DETAILS ABOUT THE VERSIONS NOT INCLUDED IN THE STUDY (TOMCAT).

Versions with only added classes	10
Versions with only removed classes	2
Version with no A/R classes	3

So, the versions retained in the study (15 in total) are those for which classes have been added and others were removed. We calculated the average value of each of the source code metrics (Qi, CBO, RFC, WMC, LCOM and LOC) for the two groups (A and R) of each version. These values are then used to compare the quality (according to the values of the selected OO metrics) of the added classes to the quality of the removed ones. We focused on coupling, cohesion, complexity ans size. Table XVII presents the results of this comparison.

TABLE XVII.
COMPARISON BETWEEN ADDED AND REMOVED CLASSES ALONG THE EVOLUTION OF TOMCAT.

	Avg Adds value > Avg Rems value	**Avg Adds value < Avg Rems value**
Qi	4	**11**
CBO	**9**	6
RFC	**10**	5
LOC	**9**	6
WMC	**10**	5
LCOM	**10**	5

From Table XVII, it can be seen that, for example, according to the Qi metric : (1) the average value of the Qi metric of the added classes of 4 versions is greater than the average value of the removed classes, and (2) the average value of the Qi metric of the added classes of 11 versions is less than the average value of the removed classes. According to the metric CBO, inversely to the metric Qi, we can see that: (1) the average value of the CBO metric of the added classes of 9 versions is greater than the average value of the removed classes, and (2) the average value of the CBO metric of the added classes of 6 versions is less than the average value of the removed classes. The same trend is observed for the other OO metrics (RFC, WMC, LCOM and LOC) in the sense that the number of versions for which the values of these metrics for the added classes are greater than those of removed classes is higher than the number of versions in which the inverse is observed.

Overall, it can be observed from the values of the OO metrics that the quality in terms of coupling, cohesion, complexity and size of added classes (about 2/3) is relatively lower than the quality of the removed ones. The values of all metrics actually all increased (added classes versus removed classes). This trend is well reflected in the values of the Qi metric. This shows clearly that the Qi metric reflects here also the evolution of the OO metrics (in terms of improvement or degradation of quality).

In order to validate these observations, we used the statistical test of Student, which is one of the most commonly used statistical significance tests applied to small data sets (populations samples). The test is used for the comparison of the means of the two groups of classes (A and R) for each of source code metrics. We applied the typical significance threshold $\alpha = 0.05$. The outcome of this test is the acceptance or rejection of the null hypothesis (H_0) we tested:

H_0 : The means are not significantly different.

If H_0 is rejected, it means that the difference between the average of the two samples is significantly different of 0 (H_1). Table XVIII presents the results of the test.

TABLE XVIII.
RESULTS OF THE STATISTICAL TEST.

	p value	**$P(RH_0 \mid H_0$ true)**
Qi	< 0.0001	< 0.01%
CBO	< 0.0001	< 0.01%
RFC	< 0.0001	< 0.01%
LOC	0.001	< 0.08%
WMC	0.001	< 0.13%
LCOM	0.115	11.54%

These results show that the two groups of added classes and removed ones differ significantly (mean values) for all of the metrics (except for LCOM). We can then reject the null hypothesis H_0. From the observations made in this section and the results of the statistical test, we can reasonably support Hypothesis 3. By validating the three hypotheses of our study, we can conclude that the Qi metric, as a synthetic metric, can be used to observe how quality (from an internal point of view) evolves along the evolution of software systems.

E. Threats to Validity

The study presented in this paper should be replicated using many other OO software systems in order to draw more general conclusions about the ability of the Qi metric to reflect the evolution of the quality (from an internal perspective) of evolving software systems. 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 three open source software systems written in Java. Even if the collected data cover, for each system, a period of several years (4 years for two systems and 7 years for the third one), we do not claim that our results can be generalized to all systems, or software systems written in other languages. So, the findings in this paper should be viewed as exploratory and indicative rather than conclusive.

Moreover, the study has been performed on open source software systems. It would be interesting to replicate the study on industrial systems. It is also possible that facts such as the development style used by the developers for developing (and maintaining) the code of the subject systems (or other related factors) may affect the results or produce different results for specific applications.

VI. Conclusions and Future Work

In this paper, we investigated the quality evolution of three open source Java software systems using metrics. Software quality was addressed from a structural point of view. We wanted to investigate if the Qi metric, a metric that we proposed in a previous work, can be used to observe how quality evolves along the evolution of released versions of the subject software systems. We used OO design metrics, specifically the Chidamber and Kemerer (CK) metrics suite, for measuring the internal quality of a released version. Empirical studies provide, indeed, evidence that there exist a relationship between these metrics and various software quality attributes.

We performed an empirical analysis using historical data collected from the successive released versions of the three subject software systems. We investigated, along the evolution of these systems, three different issues: the evolution of the Qi metric and the internal quality of the studied systems, the evolution of various size attributes, and the quality of the added classes versus the quality of removed ones. Empirical results provide evidence that the Qi metric reflects properly the quality evolution of the studied software systems.

The achieved results are, however, based on the data set we collected from only three subject systems. The findings in this paper should be viewed as exploratory and indicative rather than conclusive. They show, at least, that the Qi metric, as a synthetic metric, offers a promising potential for capturing (reflecting) the quality evolution of evolving software systems. Further investigations are, however, needed to draw more general conclusions.

As future work, we plan to: investigate if the Qi metric may be used to observe the evolution of various external software quality attributes, investigate if the Qi metric may be used to support Lehman's laws, and finally replicate the study on other OO software systems to be able to give generalized results.

Acknowledgements

This project was financially supported by NSERC (National Sciences and Engineering Research Council of Canada) and FRQNT (Fonds de Recherche du Québec – Nature et Technologies) grants.

References

[1] M. M. Lehman, and L. A. Belady, "Program Evolution: Processes of Software Change", Academic Press, 1985.

[2] I. Sommerville, "Software Engineering," 9th Edition, Addison Wesley, 2010.

[3] G. Xie, J. Chen, I. Neamtiu, "Towards a better understanding of software evolution: An Empirical study on open source software", Proceedings of the International Conference on Software Maintenance (ICSM), pp. 51-60, 2009.

[4] M. M. Lehman, "Laws of Software Evolution Revisited", Position Paper, EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, pp 108-124, 1997.

[5] M. M. Lehman, J. F. Ramil, P. D. Wernick, P. E. Perry, and W. M. Turski, "Metrics and Laws of Software Evolution–The Nineties View", Proceedings of the 4th International Software Metrics Symposium, pp. 20-32, 1997.

[6] D. L. Parnas, "Software aging", Proceedings of the 16th International Conference on Software Engineering (ICSE), pp. 279-287, 1994.

[7] J. Van Gurp, and J. Bosch, "Design Erosion: Problems & Causes", Journal of Systems and Software, vol. 61, no. 2, pp. 105-119, 2002.

[8] H. Zhang, and S. Kim, "Monitoring software quality evolution for defects", IEEE Software, vol. 27, no. 4, pp. 58-64, 2010.

[9] ISO/IEC 9126, "Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines For Their Use", 1991.

[10] Y. Lee, J. Yang, and K. H. Chang, "Metrics and evolution in open source software", 7th International Conference on Quality Software (QSIC), pp. 191-197, 2007.

[11] T. Mens, S. Demeyer, "Future Trends in Software Evolution Metrics", Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE). ACM, pp. 83-86, 2001.

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

[13] V. Basili, L. 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.

[14] N. E. Fenton, and S. L. Pfleeger, "Software Metrics : A Rigorous & Practical Approach", 2nd Edition, PWS Publishing Company, 1997.

[15] 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.

[16] M. Badri, and F. Touré, "Empirical Analysis for Investigating the Effect of Control Flow Dependencies on

Testability of Classes", Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), USA, 2011.
[17] M. Badri, and F. Toure, "Evaluating the Effect of Control Flow on the Unit Testing Effort of Classes: An Empirical Analysis", Advances in Software Engineering Journal, vol. 2012, Article ID 964064, 13 pages, 2012.
[18] M. Badri, N. Drouin, and F. Touré, "On Understanding Software Quality Evolution from a Defect Perspective: A Case Study on an Open Source Software System", Proceedings of the IEEE International Conference on Computer Systems and Industrial Informatics, Sharjah, UAE, December 18-20, 2012.
[19] N. Drouin, M. Badri, and F. Touré, "Metrics and Software Quality Evolution: A Case Study on Open Source Software", Proceedings of the 5th International Conference on Computer Science and Information Technology, Hong Kong, December 29-30, 2012.
[20] S. R. Chidamber, and C. F. Kemerer, "A Metric Suite for Object-Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.
[21] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems", Journal of Systems and Software, 51, pp. 245-273, 2000.
[22] M. Dagpinar, and J. H. Jahnke, "Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison", Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), pp. 155-164, 2003.
[23] S. Eski, and F. Buzluca, "An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes", 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 566-571, 2011.
[24] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu, "Empirical Study of Software Quality Evolution in Open Source Projects Using Agile Practices", CoRR, Vol. abs/0905.3287, 2009.
[25] 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, 2010.
[26] R. Subramanyan, and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", IEEE Transactions on Software Engineering, vol. 29, no. 4, pp. 297-310, 2003.
[27] 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.
[28] N. Nagappan, T. Ball, A. Zeller, "Mining Metrics to Predict Component Failures", Proceedings of the 28th International Conference on Software Engineering (ICSE). ACM, pp.452-461, 2006.
[29] W. Ambu, G. Concas, M. Marchesi, and S. Pinna, "Studying the evolution of quality metrics in an agile/distributed project", Extreme Programming and Agile Processes in Software Engineering, pp. 85-93, 2006.
[30] M. M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle", Journal of Systems and Software, vol. 1, no. 3, pp. 213-221, 1980.
[31] A. Jermakovics, M. Scotto, and G. Succi, "Visual Identification of Software Evolution Patterns", Proceedings of the 9th International Workshop on Principles of Software Evolution (IWPSE): in Conjunction with the 6th ESEC/FSE Joint Meeting, pp. 27-30, 2007.
[32] T. Mens, J. Fernandez-Ramil, and S. Degrandsart, "The Evolution of Eclipse", Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pp. 386-395, 2008.
[33] J. Bansiya, and C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment", IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 4-17, 2002.
[34] L. Yu, S. Ramaswamy, and A. Nail, "Using Bug Reports As a Software Quality Measure", Proceedings of the 16th International Conference on Information Quality (ICIQ), pp. 277-286, 2011.
[35] K. K. Aggarwal, Y. Singh, A. Kaur, and R. Lalhotra, "Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A Replicated Case Study", Software Process: Improvement and Practice, vol. 16, no. 1, 2009.
[36] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis", IEEE Transactions on Software Engineering, vol. 24, no. 8, pp. 629-639, 1998.

Nicholas Drouin is a student of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He finished his master in computer science (software engineering) at the University of Quebec at Trois-Rivières. His main areas of interest include object-oriented programming, software quality metrics, software evolution as well as various topics of software engineering.

Mourad Badri is a professor of computer science (software engineering) at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, software quality assurance, software maintenance and evolution as well as various topics of software engineering.

Fadel Touré is a PhD student of computer science (software engineering) at the Department of Computer Science and Software Engineering of the University of Laval, Québec. He finished his master in computer science (software engineering) at the University of Quebec at Trois-Rivières. His main areas of interest include object-oriented programming, software quality and metrics, software quality assurance as well as various topics of software engineering.