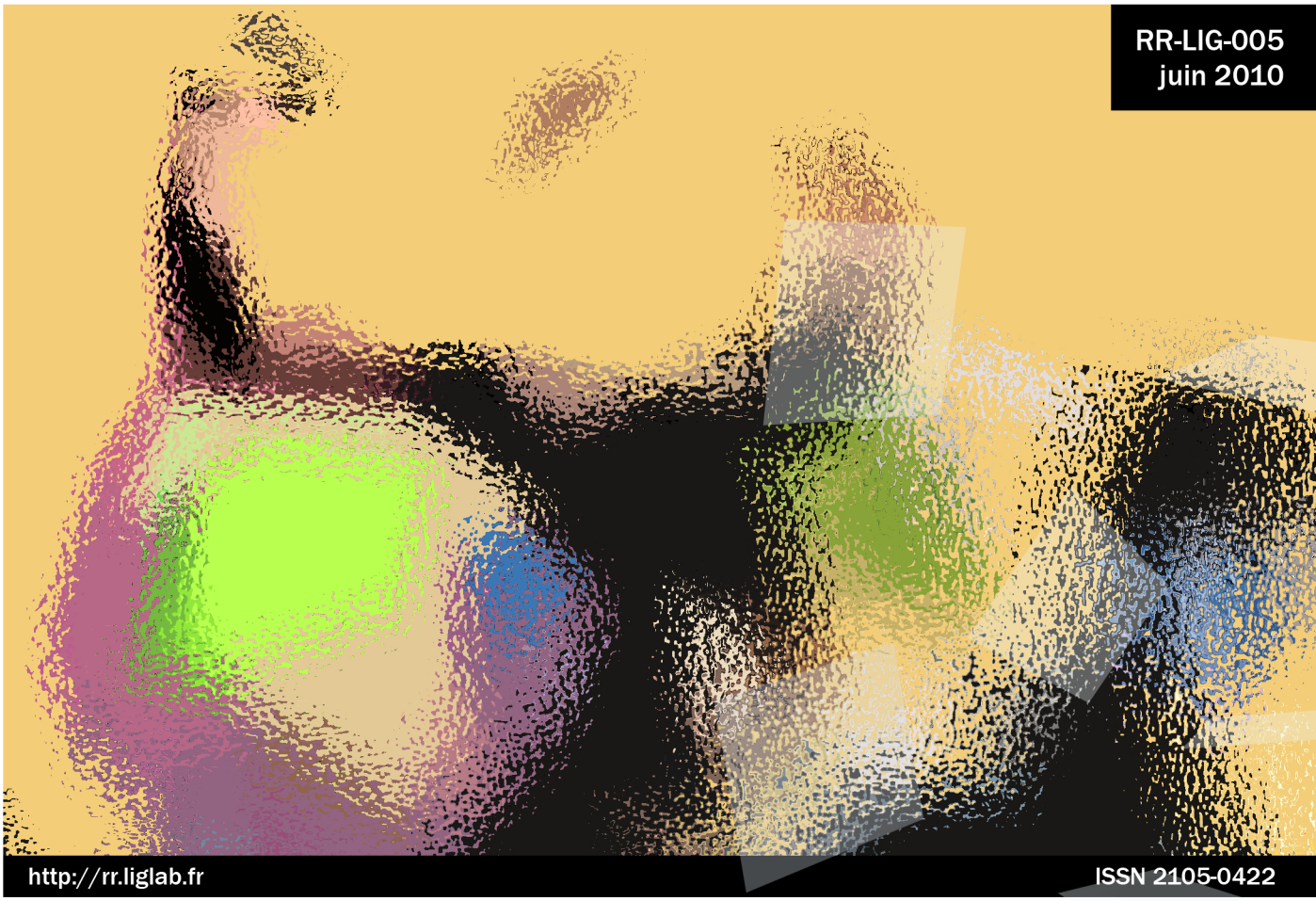
Survey of source code metrics for evaluating testability of object oriented systems

Muhammad Rabee Shaheen, Lydie Du Bousquet

Les rapports de recherche du LIG

Survey of source code metrics for evaluating testability of object oriented systems

Muhamad SHAHEEN, Drs, LIG, Grenoble University (UJF), France
Lydie DU BOUSQUET, Associated Professor, LIG, Grenoble University (UJF), France



RR-LIG-005
juin 2010

http://rr.liglab.fr
ISSN 2105-0422

L I G

Survey of source code metrics for evaluating testability of object oriented systems

Muhammad Rabee SHAHEEN
Université de Grenoble (UJF),
Laboratoire d'Informatique de Grenoble (LIG),
BP 72,
38402 Saint Martin d'Hères cedex,
France
and
Lydie du Bousquet
Université de Grenoble (UJF),
Laboratoire d'Informatique de Grenoble (LIG),
BP 72,
38402 Saint Martin d'Hères cedex,
France

Software testing is costly in terms of time and funds. Testability is a software characteristic that aims at producing systems easy to test. Several metrics have been proposed to identify the testability weaknesses. But it is sometimes difficult to be convinced that those metrics are really related with testability. This article is a critical survey of the source-code based metrics proposed in the literature for object-oriented software testability. It underlines the necessity to provide testability metrics that are proved to be intuitive and adequate for the testing cost prediction.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics

General Terms: Measurement
Additional Key Words and Phrases: software testing; software testability; testability evaluation; source-code based metrics; object oriented program analysis

1. INTRODUCTION

Software testing is the process of executing a program with the intent of finding errors [Myers 1979]. It has emerged as one of the major techniques to evaluate the implementation reliability. Unfortunately, testing is usually an expensive process. It can represent more than 40% of the total cost of the software development [Salem et al. 2004]. For this reason, being able to characterize and to produce systems easy to test (i.e. testable systems) has become a preoccupation more and more important.

Testability denotes the ability of a system to be tested. Originally, testability was defined for hardware components. In this context, testability is often characterized through observability and controllability. To test a (hardware) component, one must be able to control its inputs and observe its outputs. When a component is embedded, the presence of other components can make impossible to directly access to its inputs and outputs. The additional effort required to build test for the em-

bedded component partly depends on the architectural design. This is represented by the controllability and the observability.

For software systems, the notion of testability is becoming more and more popular because test is one of the most used way to valid a system and it is an (very) expensive process. Several studies were carried out to identify software testability characteristics. For instance, controllability and observability have been adapted for software programs, and several specific definitions of software testability were proposed. In [Bache and Mullerburg 1990], testability is defined as the effort needed for testing. For Binder, testability is the relative ease and expense of revealing software faults [Binder 1994]. Other definitions allow a quantitative evaluation of the testing effort. For IEEE, it is also considered 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" [of Electrical and Engineers 1990]. In [Bertolino and Strigini 1996], the testability is the probability that a test of the program on an input drawn from a specified probability distribution of the input is rejected, given a specified oracle and given that the program is faulty. In other words, it is the probability to observe an error at the next execution if there is a fault in the program.

From these definitions, lots of metrics have been proposed for software testability evaluation. These metrics can be evaluated at different moment in the development phase (i.e. model or code). They evaluate the complexity or the scope of testing, or both [Binder 1994]. The *scope* evaluates how many test cases have to be produced. The *complexity* indicates how much it is difficult to produce a test. For some cases, lots of test cases may be required, but it could be easy to identify them; or few tests may be required but it could be very difficult to design them.

Most of these metrics focus on source code analysis. The main reason is that testability is often evaluated at this level in the industry. It is not done for improvement purposes: it is too late (and thus too expensive) to deeply modify the system and thus its testability. Evaluating testability at the source code level is mainly done for planning and resource management. Moreover, in industry, the source code is often the first artefact enough formal to make possible an automatic analysis.

For this reason, our article provides a survey of the proposed source-based testability metrics. We focus on for object-oriented systems. This work is an extension of the survey proposed by Binder in [Binder 1994]. It presents more than 40 metrics which were declared in the literature to be testability relevant. The reason why there are so many metrics is that there are many strategies for test data selection/generation. Test data selection is based on the code or on the specification, at different phases (unit, integration and system testing), and with different purposes (among which achieving different coverage criteria). Moreover, there are different ways to understand what the *cost* of testing is. For instance, it can be the number of test cases, the size of test cases, the number of stubs, or the time spent to produce the tests.

Surprisingly, few metrics were really usable for testability evaluation. Some of them are hardly computable. Few are included in case tools. And few studies did formal or experimental validation that these metrics represent the scope or the

complexity of the test.

In the following, we first present scope-oriented metrics (section 2). Sections 3, 4 and 5 focus on complexity metrics. Section 4 is dedicated to observability and controllability related metrics, and metrics presented in section 5 are related to the probability to reveal the next error. Section 6 focuses on works carried out for validating testability metrics. Section 7 concludes and draws some perspectives.

2. SCOPE-ORIENTED TESTABILITY METRICS

2.1 Methods

Methods may be considered separately during unit-testing. Several sets of criteria defined for non object-oriented programs can then be used. They are generally defined with respect to the control-flow graph or the data-flow graphs [Myers 1979; Binder 1999; Gu et al. 1994; Ntafos 1988; Frankl and Weyuker 1988]. Classical control-flow graph criteria are statement, branch or decision, condition, MC/DC (Modified Condition/Decision Coverage), path. For data-flow graph, classical criteria are all-paths, all-du-paths, all-uses, all-c-uses, all-defs-uses, all-p-uses .

In [Bache and Mullerburg 1990] and [Yeh and Lin 1998], two *families of metrics* have been proposed to evaluate the number of elements which has to be covered with respect to the control-flow and data-flow graph testing strategies : respectively all-paths, visit-each-loop-paths, simple paths, structured, branches, statements, and p-uses, defs, uses, d-u-paths and dominating paths. By definition, those metrics predict the scope of the associated testing strategies, i.e the minimum number of required tests to reach the coverage criteria.

Similarly, the *Cyclomatic Complexity (CC)*, named also McCabe's complexity, [McCabe 1976; J. and W. 1989; Watson and McCabe 1996] is equal to the number of decision statements (or individual conditions) plus one. Mathematical analysis has shown that CC gives the recommended number of tests needed to test every decision point in a program [Watson and McCabe 1996]. Thus, it predicts the scope of the branch coverage testing strategy.

In [Bainbridge 1994], two flow graph metrics were defined axiomatically: *Number of Trails* metric which represents the number of unique simple paths through a flowgraph (path with no repeated nodes), and *Mask [k=2]* metric, which stands for "MAximal Set of K-Walks", where a k-walk is a walk through a flowgraph that visits no node of the flowgraph more than k times. Mask reflects a sequence of increasingly exhaustive loop-testing strategies. These two metrics measure the structural complexity of the code. One of the main benefits of defining these testability metrics axiomatically is that flowgraphs can be measured easily and efficiently with tools such as QUALMS [Bainbridge 1994].

2.2 Classes

The class is an essential concept in object oriented programming. A class groups attributes of an object and the operations on these attributes. Classes can also be considered during unit-testing.

Weighted Methods per Class (WMC) metric belongs to the Chidamber and Kemerer OO metrics suite [Chidamber and Kemerer 1994]. For a class C with methods

$M_1, M_2 ... M_n$, let $c_1, c_2, .., c_n$ be the complexity of these methods.

$$WMC = \sum_{i=1}^{i=n} c_i$$

Complexity was deliberately not defined in the original paper in order to allow a general application of this metric. If all method complexities are considered to be unity, then $\mathrm{WMC}_1 = n$ represents the number of methods. WMC_1 can be used to evaluate the number of test cases to achieve the method coverage [Binder 1994]. This criterion is one of the simplest OO code-coverage coverage criteria (corresponding to function coverage). It requires each method to be executed at least once. A similar metric to WMC_1 is *Number Of Methods (NOM)* representing explicitly the number of methods of a class [Binder 1994].

When cyclomatic complexity is used as complexity measure to compute WMC (WMC_{CC}), it evaluates the number of the test cases required to test all the methods of the class to reach the decision coverage criteria.

2.3 Stubs

Stubs may be required during unit or integration testing. A stub is an extra routine that is provided by the tester, to imitate another part of the system.

The *Fan Out (FOUT)* of method A is the number of local flows from method A plus the number of data structures which A updates [Henry and Kafura 1981]. In other words FOUT estimates the number of *methods* to be stubbed, to carry out a unit testing of method A

Binder proposes to use *Response For Class (RFC)* metric to evaluate how many stubs has to be produced for unit testing at class level [Binder 1994]. RFC is one of the Chidamber and Kemerer metrics suite [Chidamber and Kemerer 1994]. It is defined as the count of the methods defined in a class, in addition to the methods that are called directly by a method of this class. The number of methods to be stubbed corresponds to the number of calls of methods defined outside the class/subsystem under test. Since RFC counts also the number of methods defined within the class, RFC only provides an approximation of the number of *methods* to be stubbed, to carry out a unit testing of a class.

$$RFC = |RS| \quad where \quad RS = \{M\} \cup_{\forall i} \{R_i\}$$

where $\{R_i\}$ is the set of methods called by method i, and $\{M\}$ is a set of all methods in the class.

A class is coupled to another if one of them acts on the other, i.e. a method of a class uses methods or instance variables of the other. There are several definitions for the coupling between objects. One of them was proposed by Chidamber and Kemerer in [Chidamber and Kemerer 1994]: *Coupling Between Objects (CBO)* of a class is the number of other classes to which it is coupled. CBO can be used to evaluate the number of *classes* to be stubbed, in order to carry out a unit testing

of a class [Binder 1994].
Eight different levels of coupling were ordered by Jones [Page-Jones 1988]. These levels influence the different quality factors of a unit such as reusability, maintainability, understandability...etc. An extension to these levels was made to become 12 levels [Jin and Offutt 1998]. These levels evaluate the software system designs complexity, and a relatioship between these levels and the number of faults [Troy and Zweben 1993].

Class Fan Out (Class FOUT) represents the number of classes on which a given class depends [Schroeder 1999]. This metric could be used to estimate the number of classes to be stubbed.
When some strategies are used to schedule intelligently the test of the different classes in order to decrease the number of subs required to be produced, CBO or Class FOUT may be not relevant. In [Kung et al. 1995], authors show that 400 stubs would be required to test 122 classes individually (without any strategy) against 8 when an optimal test order is used. In [Jungmayr 2002], S. Jungmayr proposes a testability metric in the context of static dependencies within object-oriented systems. A dependency of a component A on a component B exists if A requires B to compile or to function correctly. If A inherits from B or if it uses method(s) or attribute(s) of B then A depends on B. Dependency relation is transitive. A dependency graph may contain cycles. Such cycles can be broken by removing some dependencies. The set of removed dependencies are called *Feedback Dependency Set*.

Number of Stubs needed to Break Cycles (NSBC) evaluates the number of stubs required to be built with an integration testing strategy.

$$NSBC = |C_{Fb}|$$

where C is the set of all components, C_{Fb} a feedback component set ($C_{Fb} \subset D$), and D is the set of all dependencies. Finding a smallest feedback component set is NP-complete. To identify a small feedback component set S. Jungmayr proposed an algorithm based on both Tarjan and greedy algorithms [Jungmayr 2002].

2.4 Inheritance

Inheritance is one of the main features of object-oriented programming paradigm. Since it has been demonstrated that inheritance may be abused in many ways [Armstrong and Mitchell 1994], one may expect that several testing criteria would have been dedicated to inheritance testing. Surprisingly, few testing methods/criteria deal with inheritance [Harrold et al. 1992; Perry and Kaiser 1990; Cheatham and Mellinger 1990; Fiedler 1989; Chung et al. 1997]. Most of them restrict testing to validate changes in the inherited features (methods and attributes).

In [Binder 1999], all inherited methods should be retested. In [Harrold et al. 1992; Perry and Kaiser 1990; Cheatham and Mellinger 1990; Fiedler 1989], it is suggested to re-test only (modified) inherited features (attributes or methods). Since, the number of inherited methods is considered to be generally proportional to the

Depth of Inheritance Tree (DIT) [Chidamber and Kemerer 1994; Binder 1994], DIT is considered as a way to estimate the testing effort. But it does *not* provide an estimation of *how many* test cases have to be produced [Shaheen and du Bousquet 2008]. A class with a small inheritance tree may have more inherited methods than a class with a large inheritance tree.

When a class inherits the same property of an ancestor via multiple paths in the hierarchy, there is repeated inheritance. Repeated inheritance is not allowed in several OO languages, such as Java, C#, or VB .Net. Overuse of repeated inheritance increases software error [Chung et al. 1997]. That's why C.-M. Chung *et al.* propose a testing method to search for errors caused by the repeated inheritance. Each class concerned by repeated inheritance has to be tested in the context of its inheritance sub-trees.
Authors introduce the notion of URI (Unit Repeated Inheritance) as a specific inheritance sub-graph, where the number of nodes equals the number of edges ($G = (V, E)$ where $|V| = |E|$). Repeated inheritance tree can be decomposed as a set of basic $URIs$. For an inheritance tree, let t be the number of terminal classes (classes with no out-edges) and U_i be the set of URI related to the terminal class i. The *complexity of the repeated inheritance* is defined as $|\cup_{i=1}^{t} U_i|$. This complexity corresponds to the number of repeated inheritance sub-tree to be examined. Here again, it does not predict how many test cases have to be produced for each of them.

3. GENERAL COMPLEXITY METRICS

As we have seen previously, the Cyclomatic Complexity (CC) gives the recommended number of tests needed to test every decision point in a program [Watson and McCabe 1996]. Thus, it predicts the scope of the branch coverage testing strategy. It is also considered as an indication of the complexity of testing. Indeed, a method with a CC greater than 50 is considered to be untestable [1].
By extension, WMC_{CC} could also be considered as an indication how difficult it is to test the class. However, it could be difficult to interpret: WMC_{CC} indicates that a class A with 60 very simple methods ($c_i = 1$) will require more testing than a class B with one method having a complexity of 50. When analyzing CC for each method, B will be more difficult to test (since a method with a complexity of 50 is supposed to be untestable) and 60 simple methods will require 60 simple tests. The authors in [Michura and Capretz 2005] proposes to use WMC_{CC} with three other metrics: *Mean Method Complexity (MMC)*, *Standard Deviation Method Complexity (SDMC)* and *Number of Trivial Methods (NTM)*. Using the 4 metrics as opposed to WMC_{CC} alone allows distinguishing between certain types of classes and therefore interprets the results accordingly.

3.1 Cohesion

Cohesion is an extension to the definition of *similarity* which was proposed by Bunge [Chidamber and Kemerer 1994]. The similarity $\sigma()$ of two things is the intersection

[1] `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`

of the sets of properties of the two things.

$$\sigma(X, Y) = P(X) \cap P(Y)$$

Cohesion has been considered to be a factor influence the cost of testing [Chidamber and Kemerer 1994]. Having lack of cohesion in a class C means that this class has many functionalities which are not related to each other, and as a result the class C will behave in less predictable way than a class of a fewer functionalities.

Lack of Cohesion of a Method (LCOM) is the 6th measure of the Chidamber and Kemerer metrics suite [Chidamber and Kemerer 1994]. LCOM measures the degree of similarity between the class's methods. The more methods share the same attributes, the larger is cohesion. LCOM metric has been redefined several times [Briand et al. 1998]. We give here the definition of LCOM4, given by Hitz and Montazer. Let $G_x = (V, E)$ be a graph representing calls between class methods. V is the set of vertices, which represents the method names. E is the set of edges. $E = \{(m, n) \in V \times V | \exists i \in I_x : (m \text{ accesses } i) \wedge (n \text{ accesses } i) \wedge (m \text{ calls } n) \wedge (n \text{ calls } m)\}$ where I_x is the set of the attributes. $LCOM_4 = |E|$
LCOM has been proposed as a testability metric by Binder in [Binder 1994], because high LCOM means more states that have to be tested to prove the absence of side effect among the methods.

Tight Class Cohesion (TCC) is the percentage of pairs of public methods of the class which are directly connected [Bieman and Kang 1995].

$$TCC = NDC/NP \quad where \quad NP = N * (N-1)/2$$

where:

—N the number of methods

—NP represents the number of possible connections

—NDC number of indirect connections

Loose Class Cohesion (LCC) considers the pairs connected directly or indirectly [Bieman and Kang 1995].

$$LCC = (NDC + NIC)/NP$$

where NIC represents the number of indirect connections.

Information CoHesion (ICH) was proposed by Lee et al. [Lee et al. 1995]. It is based on the information flow. It considers the cohesion of a method m implemented in a class c as the number of the invocations to other non-inherited methods of class c, weighted by the number of the parameters of the invoked methods. Cohesion between calling and called methods is stronger if the latter has more parameters more information is passed.

Most Cohesive Component (MCC) is introduced in [Seok Chae and Rae Kwon 1998] as the most cohesive form if each class's method has interaction with all of instance variables (special methods such as *set* and *get* are excluded).

3.2 Polymorphism

It is an important feature in OO programming, defined as a characteristic of being able to have multiple forms. To test software that uses polymorphism, one should test all possible bindings of receiver classes and target methods at different call points. The authors in [Lin and Huang 1998] define a *testability of polymorphism* metric in inheritance hierarchy, based on the descendant paths.
Also a *polymorphism factor (PF or POF)* was defined to represent the actual number of possible different polymorphic situations[e Abreu and Melo 1996]. A very high POF value (above 10%) will reduce the benefits of polymorphism.

Other metrics have been proposed in [Binder 1994] to measure the complexity as a result of the polymorphism such as *percent of Dynamic calls (DYN)*, *percent of non-over loaded calls (OVR)*, *number of yo-yo paths visible to CUT (Bounce-C)*, and *number of yo-yo paths in SUT (Bounce-S)*. Although these metrics have not been evaluated [Binder 1994], they were proposed as an indicator of the opportunities for faults.

3.3 Inheritance

Inheritance influences the scope of testing as previously indicated. Binder indicates that it also influences the complexity of testing. Like DIT, several other measures were proposed for inheritance tree complexity evaluation. *Class Fan-In (FIN)* is the number of parent classes of a subclass. It is applied only in multiple inheritance languages. A high Class FIN value increases the possibility of incorrect bindings [Binder 1994].

Number Of Children (NOC) is the number of classes that inherit directly from a class [Chidamber and Kemerer 1994]. It is the number of immediate subclasses subordinated to a class in the class hierarchy. It indicates how many derived classes will be affected by some modification in the parent class. So if a modification in the parent class affects the derived classes, it is required to retest the methods in the children. The higher NOC is, the more tests should be produced.

In [Brito E Abreu and Carapua 1994], authors propose a set of measures called MOOD set, for object-oriented design quality evaluation. This set includes *Method Inheritance Factor (MIF)* and *Attribute Inheritance Factor (AIF)*. They compute the number of the inherited methods/attributes in all classes divided by the number of available methods/attributes (inherited and defined) for all classes. Those factors are defined to evaluate the inheritance complexity of the whole system. They could be adapted to evaluate MIF and the AIF for each class (instead of for all classes together).

3.4 Encapsulation

Encapsulation is defined as *"a software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module"* [IEE 1990]. Using the encapsulation makes private data inaccessible directly during testing. An *encapsulation factor (EF)* metric for a class was proposed to measure the encapsulation level of a class. EF was defined as a function of two parameters *privacy* and *unity*, where privacy is based on the private data members (the data visibility),

and unity is the cohesion between the attributes and methods [Saini and Aggarwal 2007].

4. COMPLEXITY METRICS RELATED TO OBSERVABILITY AND CONTROLLABILITY

Originally, testability was defined for hardware components. In this context, testability is often characterized through observability and controllability. To test a component, one must be able to control its inputs and observe its outputs. When a component is embedded, its controllability and observability can be decreased. This partly depends on the architectural design. Observability and controllability notions were adapted from hardware to software systems.

4.1 Domain Testability

Two *Domain Testability* metrics have been proposed by Freedman for non-OO software and based on these concepts [Freedman 1991]. A procedure F is observable if distinct outputs are generated from distinct inputs, and an expression procedure F is controllable if the set of all evaluations of F covers all values in the range (co-domain of F). Observability and controllability *extensions* are the input/output variables required to achieve the definitions of observability and controllability.
The *observability (Ob)* and the *controllability (Ct)* according to the domain size of the added inputs/outputs.

$$Ob = log_2(|ID_1| * .... * |ID_n|)$$

$$Ct = log_2(|OD_1| * .... * |OD_m|)$$

where ID_i is the domain of the ith added input and OD_j is the domain of the jth added output.
A limit of these metrics is that the cardinality of certain types cannot be calculated (i.e. vector, array, object, etc). Moreover, one important feature in object-oriented paradigm is that objects preserve states. So attributes could be used as implicit input/output for a method. Therefore, the notion of observability an controllability extensions have to be adapted to OO features.

4.2 Observability and Controllability of Component

In [Washizaki et al. 2003], five metrics were proposed as reusability measure of software component. Two of them are related to observability and one to controllability. The *Rate of Component Observability (RCO)* represents the percentage of readable attributes in all fields implemented within the Façade class of a component. If the value of RCO is in [0.17, 0.42] (confidence interval) the height of the observability is supposed to be appropriate [Washizaki et al. 2003]. It is a variation of *percent Public And Protected (PAP)* defined in [Binder 1994].

$$RCO(c) = P_r(c)/A(c) \quad when(A(c) > 0), \quad 0 \ otherwise$$

where $P_r(c)$ is the number of readable properties in the component c, and $A(c)$ is the number of fields in c's Façade class.

The *Self-Completeness of Component's Return value (SCCr)* is the percentage of business methods without any return value in all business methods implemented within a component c. It can be extended to non business methods and allows to detect the absence of return values, which decreases observability. However, observability is not limited to the existence of return values.

$$SCCr(c) = B_v(c)/B(c) \quad when(B(c) > 0), \quad 1 \ otherwise$$

where $B_v(c)$ is the number of business methods without return value in c, and $B(c)$ the number of all business methods in c.

The *Self completeness of component's parameter (SCCp)* is the percentage of business methods without any parameters in all business methods implemented within a component c. SCCp can be extended to other application layers (i.e. non business methods). Since it can be easier to test methods which have no input parameter (see Category and Partition [Ntafos 1998]), classes with high SCCp may be easier to test than some with low SCCp. One limit of SCCp is that it is independent from the difficulty to test methods with parameters. Let A be a class of several methods each of them has one parameter, $(SCCp(A) = 0)$. Let B be a class of two methods, one with several methods and the other without any parameter, $(SCCp(B) > 0)$. A is probably easier to test than B. Moreover $SCCp$ does not capture the situation where methods have implicit parameters (attributes).

$$SCCp(c) = B_p(c)/B(c) \quad when(B(c) > 0), \quad 1 \ otherwise$$

where B_p is the number of business methods without parameters in c.

4.3 System Testability - STA

System Testability (STA) of an object-oriented software is defined as mathematical mean of all the objects testability obtained in the system [Wang et al. 1997].

$$STA = \frac{1}{n} \sum_{j=1}^{m} OTA_j$$

where OTA_j is the object testability defined as the product of its test controllability and observability. Controllability of an object is the ability to control all the basic control structures within the object. Observability of an object is the ability to indicate the values of any variables within the path(s) sensitised by the current test case.

STA is the average of its component testability values. This definition does not take into account the architecture of the system, which is quite unusual. It is usually expected that the architecture influences the observability and the controllability.

5. TESTABILITY METRICS RELATED TO ERROR LIKELIHOOD

Testing aims at finding errors [Myers 1979]. The easier it is to find errors, the easier testing is. One definition of software testability given by A. Bertolino is "the probability that a test of the program on an input drawn from a specified probability distribution of the input is rejected, given a specified oracle and given that the program is faulty" [Bertolino and Strigini 1996]. In other words, it is the probability to observe an error at the next execution if there is a fault in the program. Several metrics are related to this definition.

5.1 Propagation Infection Execution - PIE

Propagation Infection Execution (PIE) analysis is a well-known metric proposed for testability. It has been proposed by Voas [Voas 1992; Voas and Miller 1995]. PIE measure aims at computing the sensitivity of individual locations in a program.
The sensitivity of a program location refers to the minimum likelihood that a fault at that location will produce incorrect output, under a specified input distribution. This measure has its origin in the RELAY model used for error detection [Morell 1990]. It relies on the fact that for discovering a fault in a program, three conditions should be met. First the statement that contains the fault should be executed. Secondly the state of the variable should be infected. And at last, it should be propagated to an output.
Testability of a software statement $T(s) = Re(s) * Ri(s) * Rp(s)$ where $Re(s)$ is the probability of the statement execution, $Ri(s)$ the probability of internal state infection and $Rp(s)$ the probability of error propagation.
PIE analysis determines the probability of each fault to be revealed. PIE original metric requires sophisticated calculations. It does not cover object-oriented features such as encapsulation, inheritance, polymorphism, etc.
In [Lo and Shi 1998], authors propose an adaptation of PIE analysis to compute the *testability of a class t(C)*. Based on t(C), the testability of the class is derived with respect to different factors: cohesion, communication and inheritance.

5.2 Domain-Range Ratio - DRR and Visibility Component - VC

A simplification of sensitivity analysis has been proposed with the *Domain-Range Ratio (DRR)*. DRR of a specification is the ratio between the cardinality of the domain to the cardinality of the range. DRR depends only on the number of values in the domain and the range, not on the relative probabilities that individual elements may appear in these sets [Voas and Miller 1993].
For a program, when the input domain is larger than the output domain, information about the internal states may not be communicated in the outputs. This information may have included evidence that internal states were incorrect. This loss suggest a lower testability.
For instance, let us consider two functions $F(x) = \mathrm{x}\ mod\ 2$ and $G(x) = 2 * x$. F has a domain on all R and a range on {0,1}. For such function, it is difficult to predict if the result is really the one that corresponds exactly to the given input: it has an unlimited and infinite set of inputs produce the same result (output). G has a domain on all R, and the range also on all R. Any two different inputs will produce two different outputs. Detecting an error for $G(x)$ is easier than detecting an error on $F(x)$.
DRR evaluates how much an application is supposed to hide faults. It is a priori information, which can be considered as a rough approximation of testability.

J. McGregor and S. Srinivas proposed an extension of DRR for object-oriented programs called *Visibility Component (VC)* [McGregor and Srinivas 1996]. VC is the cardinality of possible outputs (including the exceptions) divided by the cardinality of possible inputs. Two types of parameters for a method are considered: implicit and explicit ones.
Like domain testability measures, cardinality of certain types cannot be calculated

(i.e. vector, array, object, etc) by DRR or VC. This is major drawback makes impossible to use DRR or VC practically.

6. VALIDATION OF THE METRICS

In [Kaner and Bond 2004], C. Kaner and W.P. Bond question the *construct validity* of software engineering metrics, as several other authors before them. The general question is "*how we know that we are measuring the attribute that we think we are measuring?*" Same question can be asked for all the metrics presented here: how we know that those metrics really correspond to testability?

Several frameworks or methodologies were proposed in the literature for measurement and metrics validation [237 1993; Kitchenham et al. 1995; Sheppard and Kaufman 2001; Schneidewind 1992; Mendonça and Basili 2000; Harrison et al. 1998; Weyuker 1988]. Among them, the standard IEEE 1061 [237 1993] proposes a methodology to define metrics where the fifth step is Validation. *Predictive metrics results are compared to the direct metrics results to determine whether the predictive metrics accurately measure their associated quality factor.* Moreover, the standard lays out six validation criteria: correlation, tracking, consistency, predictability, discriminative power and reliability. Another draft standard for testability and diagnosability characteristics and metrics was developed by (D&MC), a subcommittee of IEEE [Sheppard and Kaufman 2001], the purpose of this standard was to provide formal and unambiguous definition of testability, where this definition should be independent of specific test, diagnosis process and system under test.

In [Mouchawrab et al. 2005], the authors introduced a generic measurement framework for OO software testability, which is based on a theory expressed as a set of operational hypotheses. They identified 20 hypotheses, 4 of these hypotheses are related to inheritance concept (i.e inherited features, operation rule...etc), which could be identified, more or less, as a result of the metric DIT. Other hypotheses are related to number of paths, coupling, dependency cycles, cohesion, complexity of pre/post conditions and invariants...etc. Although this framework is introduced for a high level design, the most of these hypotheses correspond to certain metrics presented in Table I. A future work could be done to associate formally these hypotheses to existed metrics.

A large amount of empirical studies have been carried out to establish the relation between OO metrics and fault-proneness of classes [Basili et al. 1996; Briand et al. 1999; Tang et al. 1999; Briand et al. 2000; Briand et al. 2001; Emam et al. 2001; Yu et al. 2002; Gyimóthy et al. 2005; Zhou and Leung 2006]. Classical metrics such as WMC, LCOM, DIT, RFC, and CBO were considerate. But the studies were not correlated to testability.

In [Bruntink and van Deursen 2006], authors have evaluated the correlation between a set of OO source code metrics (among which DIT) and their capabilities to predict the effort needed for testing, expressed as dLOCC (*Lines Of Code for Class*) and dNOTC (*Number of Test Cases*). Somewhat surprisingly, DIT was not correlated to dNOTC. This was explained by the fact that inherited methods were probably not systematically re-tested. However, if all inherited methods are re-tested, it was expected that the number of test cases should increase with respect to DIT, as it has been shown after in [Shaheen and du Bousquet 2008].

The validation of testability is a hard work, especially for the complexity metrics. The main difficulty relies on the fact that the *effort* to test is subjective and depends on the point of view. For instance, let us consider the *Rate of Component Observability* (RCO) [Washizaki et al. 2003] and the *percent Public And Protected* (PAP) [Binder 1994]. They both represent the percentage of readable attributes, but are used differently with respect to the testability analysis: if a high RCO is supposed to ease testing because observability is increased [Washizaki et al. 2003], a high PAP is supposed to increase the difficulty of testing, because there are more opportunities for side effects. A challenge is thus to propose some definition(s) for what could be the effort to test.

The validation of scope metrics seems to be easier. By definition, scope metrics predict the number of tests to produce with respect to a testing approach (see Table I). To validate those metrics empirically, one can compare the expected number of test cases given by the metric and the effective one when applying the testing method. The possible differences can be due to the impossibility to reach the associated coverage criteria because of the infeasible execution paths for instance. If empirically, a scope metric prediction is very different from the effective number of test cases, it may be error-prone and thus should not be used.

7. CONCLUSION AND PERSPECTIVES

Since testing is expensive, predicting testability is important to organize test and/or to build easy-to-test systems. However, software testability is a concept difficult to capture and formalize. Lots of factors can effect testability [Mouchawrab et al. 2005; Binder 1994]. Lots of metrics have been proposed to predict testability, and especially at the source code level. Evaluating testability at the source-code level helps testers to detect parts of the system supposed to be difficult to test. It allows organizing and planning testing work

In this paper, we have collected more than 40 metrics, classified in scope and complexity testability metrics. Scope metrics predict how many tests cases are required by some testing strategies. Complexity metrics predict how difficult it will be to generate test cases. The complexity can be evaluated with respect to the structure of the system, the observability and the controllability attributes, or the probability to find the next error. Our work is an extension of the one done by Binder in [Binder 1994], which presents 21 metrics which can be evaluated on code.

It is really surprising to notice how few of these metrics are really usable. Less than one half of them are implemented in classical metrics tools (see Table II). Moreover testability metrics have been rarely validated neither formally nor empirically.

As a perspective for software testability, if we want to help practitioner to produce easy-to-test software programs, we should be able to offer a validated, meaningful and easy-to-understand set of metrics. From this survey, we are hardly convinced that a *set* of metrics is necessary, because (1) existing metrics are too ambitious by trying to obtain a large amount of information through a single number [Michura and Capretz 2005] and (2) there are many different testing approaches and ways to evaluate the testing cost.

	Metrics	Effort estimation	Available tools
1	Control-flow graph metric suite [Bache and Mullerburg 1990]	# of TC for control-flow graph coverage	no
2	Data-flow graph metric suite [Yeh and Lin 1998]	# of TC for data-flow graph coverage	no
3	CC [McCabe 1976; J. and W. 1989; Watson and McCabe 1996]	# of TC for branch coverage (decision point)	yes
4	WMC_1 [Chidamber and Kemerer 1994; Binder 1994]	# of TC to achieve the method coverage	yes
5	NOM [Binder 1994]	# of TC to achieve the method coverage	yes
6	WMC_{CC} [Chidamber and Kemerer 1994; Binder 1994]	# of TC to achieve the branch coverage at the class level	yes
7	RFC [Chidamber and Kemerer 1994; Binder 1994]	# of methods to be stubbed	yes
8	CBO [Chidamber and Kemerer 1994; Binder 1994]	# of classes to be stubbed (no integration strategy)	yes
9	FOUT [Henry and Kafura 1981]	# of methods to be stubbed (no integration strategy)	yes
10	NSBC [Jungmayr 2002]	# of classes to be stubbed (with an integration strategy)	no
11	DIT [Chidamber and Kemerer 1994; Binder 1994]	proportional to # of TC (when inheritance is re-tested)	yes
12	Complexity of the repeated inheritance [Chung et al. 1997]	# of repeated inheritance sub-tree to examine	no

Table I. Scope source-code testability-related metrics

	Tool name	Reference	Calculated metrics
1	CKJM	http://www.spinellis.gr/sw/ckjm/	WMC, DIT, NOC, CBO, RFC, LCOM,...
2	NetBeans Metrics Module	http://metrics.netbeans.org/	WMC, CBO, RFC, DIT, NOC,...
3	Eclipse Metrics plug-in	http://metrics.sourceforge.net/	LCOM, WMC, CC, DIT,...
4	JStyle	http://www.mmsindia.com/jstyle.html	RFC, LCOM, Fan-In, Fan-Out, WMC, DIT
5	Understand for Java	http://www.scitools.com	LCOM, DIT, CBO, NOC, RFC,...

Table II. Metrics Tools

REFERENCES

12 Mar 1993. IEEE standard for a software quality metrics methodology. *IEEE Std 1061-1992*.

28 Sept. 1990. IEEE standard Glossary of Software Engineering Terminology.

Armstrong, J. and Mitchell, R. 1994. Uses and abuses of inheritance. *Software Engineering Journal 9,* 1 (january), 19–26.

Bache, R. and Mullerburg, M. 1990. Measures of testability as a basis for quality assurance. *Software Engineering Journal 5,* 2, 86–92.

Bainbridge, J. 1994. Defining testability metrics axiomatically. *Softw. Test., Verif. Reliab. 4,* 2, 63–80.

Basili, V. R., Briand, L. C., and Melo, W. L. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng. 22,* 10, 751–761.

Bertolino, A. and Strigini, L. 1996. On the use of testability measures for dependability assessment. *IEEE Trans. Software Eng. 22,* 2, 97–108.

Bieman, J. M. and Kang, B.-K. 1995. Cohesion and reuse in an object-oriented system. In *SSR*. 259–262.

Binder, R. V. 1994. Design for testability in object-oriented systems. *Communications of the ACM 37,* 9 (Sept.), 87–101.

Binder, R. V. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. The Addison-Wesley Object Technology Series.

Briand, L. C., Daly, J. W., and Wüst, J. 1998. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering 3,* 1, 65–117.

Briand, L. C., Wüst, J., Daly, J. W., and Porter, D. V. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software 51,* 3, 245–273.

Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H. 1999. Investigating quality factors in object-oriented designs: An industrial case study. In *ICSE*. 345–354.

Briand, L. C., Wüst, J., and Lounis, H. 2001. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering 6,* 1, 11–58.

Brito E Abreu, F. and Carapua, R. 1994. Object-oriented software engineering: Measuring and controlling the development process. In *4th International Conference on Software Quality (ASQC)*. McLean, VA, USA.

Bruntink, M. and van Deursen, A. 2006. An empirical study into class testability. *Journal of Systems and Software 79,* 9 (September), 1219–1232.

Cheatham, T. J. and Mellinger, L. 1990. Testing object-oriented software systems. In *ACM Conference on Computer Science*. 161–165.

Chidamber, S. R. and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Trans. Software Eng. 20,* 6, 476–493.

Chung, C.-M., Shih, T. K., Wang, C.-C., and Lee, M.-C. 1997. Integration object-oriented software testing and metrics. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 7,* 1, 125–144.

e Abreu, F. B. and Melo, W. L. 1996. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*. 90–99.

Emam, K. E., Benlarbi, S., Goel, N., and Rai, S. N. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng. 27,* 7, 630–650.

Fiedler, S. P. 1989. Object-oriented unit testing. *Hewlett-Packard Journal 40,* 2 (April), 69–75.

Frankl, P. G. and Weyuker, E. J. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng. 14,* 10, 1483–1498.

Freedman, R. S. 1991. Testability of software components. *IEEE Trans. Software Eng. 17,* 6, 553–564.

Gu, D., Zhong, Y., and Ali, S. 1994. On testing of classes in object-oriented programs. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, Toronto, Ontario, Canada, 22.

Gyimóthy, T., Ferenc, R., and Siket, I. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng. 31,* 10, 897–910.

Harrison, R., Counsell, S. J., and Nithi, R. V. 1998. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. Softw. Eng. 24,* 6, 491–496.

Harrold, M., McGregor, J., and Fitzpatrick, K. 1992. Incremental testing of object-oriented class structures. In *International Conference on Software Engineering (ICSE)*.

Henry, S. and Kafura, D. Sept. 1981. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on SE-7,* 5, 510–518.

J., M. T. and W., B. C. 1989. Design complexity measurement and testing. *Communication of the ACM 32,* 12, 1415–1425.

Jin, Z. and Offutt, A. J. 1998. Coupling-based criteria for integration testing. *Softw. Test., Verif. Reliab. 8,* 3, 133–154.

Jungmayr, S. 2002. Identifying test-critical dependencies. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, Montreal, Quebec, Canada, 404–413.

Kaner, C. and Bond, W. P. 2004. Software engineering metrics: What do they measure and how do we know? In *10th IEEE International Software Metrics Symposium (METRICS 2004)*. IEEE, Chicago, USA.

Kitchenham, B., Pfleeger, S. L., and Fenton, N. E. 1995. Towards a framework for software measurement validation. *IEEE Trans. Software Eng. 21,* 12, 929–943.

Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. 1995. A test strategy for object-oriented programs. In *19th International Computer Software and Applications Conference (COMPSAC'95)*. IEEE Computer Society, Dallas, Texas, USA, 239–244.

Lee, Y., Liang, B., Wu, S., and Wang, F. 1995. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *International Conference on software quality (ICSQ'95)*. Maribor, Slovenia, 81–90.

Lin, J.-C. and Huang, Y.-L. 1998. A new method for estimating the testability of polymorphism in class hierarchy. *Int. Computer Symposium*.

Lo, B. and Shi, H. 26-29 Jan 1998. A preliminary testability model for object-oriented software. *Software Engineering: Education and Practice, 1998. Proceedings. 1998 International Conference*, 330–337.

McCabe, T. J. 1976. A complexity measure. *IEEE Trans. Software Eng. 2,* 4, 308–320.

McGregor, J. D. and Srinivas, S. 1996. A measure of testing effort. In *Second USENIX Conference on Object-Oriented Technologies (COOTS)*.

Mendonça, M. G. and Basili, V. R. 2000. Validation of an approach for improving existing measurement frameworks. *IEEE Trans. Softw. Eng. 26,* 6, 484–499.

Michura, J. and Capretz, M. 2005. Metrics suite for class complexity. In *International Conference on Information Technology: Coding and Computing (ITCC'05)*. Vol. 2.

Morell, L. J. 1990. A theory of fault-based testing. *IEEE Trans. Software Eng. 16,* 8, 844–857.

Mouchawrab, S., Briand, L. C., and Labiche, Y. 2005. A measurement framework for object-oriented software testability. *Information & Software Technology 47,* 15, 979–997.

Myers, G. 1979. *The Art Of Software Testing.* Wiley-Interscience.

Ntafos, S. 1988. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 868–874.

Ntafos, S. 1998. On random and partition testing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis.* ACM Press, 42–48.

of Electrical, I. and Engineers, E. 1990. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Tech. rep., IEEE, New York, USA.

Page-Jones, M. 1988. *The practical guide to structured systems design: 2nd edition.* Yourdon Press, Upper Saddle River, NJ, USA.

Perry, D. E. and Kaiser, G. E. 1990. Adequate testing and object-oriented programming. *Journal of Object Oriented Programming 2,* 5, 13–19.

Saini, S. and Aggarwal, M. 2007. Enhancing mood metrics using encapsulation. In *ICAI'07: Proceedings of the 8th Conference on 8th WSEAS International Conference on Automation*

and Information. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 252–257.

Salem, A. M., Rekab, K., and Whittaker, J. A. 2004. Prediction of software failures through logistic regression. *Information & Software Technology 46,* 12, 781–789.

Schneidewind, N. May 1992. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on 18,* 5, 410–422.

Schroeder, M. Nov.-Dec. 1999. A practical guide to object-oriented metrics. *IT Professional 1*, 30–36.

Seok Chae, H. and Rae Kwon, Y. 1998. A cohesion measure for classes in object-oriented systems. In *5th IEEE International Software Metrics Symposium (METRICS)*. IEEE Computer Society, Bethesda, Maryland, USA, 158–166.

Shaheen, M.-R. and du Bousquet, L. 2008. Relation between depth of inheritance tree and number of methods to test. In *1st International Conference on Software Testing, Verification and Validation*. IEEE, Lillehammer, Norway.

Sheppard, J. and Kaufman, M. 2001. Formal specification of testability metrics in ieee p1522. *AUTOTESTCON Proceedings, 2001. IEEE Systems Readiness Technology Conference*, 71–82.

Tang, M.-H., Kao, M.-H., and Chen, M.-H. 1999. An empirical study on object-oriented metrics. In *6th IEEE International Software Metrics Symposium (METRICS'99)*. IEEE Computer Society, Boca Raton, FL, USA, 242–249.

Troy, D. A. and Zweben, S. H. 1993. Measuring the quality of structured designs. 214–226.

Voas, J. 1992. PIE: A dynamic Failure-Based Technique. *IEEE Transaction on Software Engineering 18,* 8 (August), 41–48.

Voas, J. and Miller, K. 1993. Semantic Metrics for Software Testability. *J. Systems Software 20*, 207–216.

Voas, J. and Miller, K. 1995. Software Testability : the new Verification. *IEEE Software 3*, 17–28.

Wang, Y., King, G., Court, I., Ross, M., and Staples, G. 1997. On testable object-oriented programming. *SIGSOFT Softw. Eng. Notes 22,* 4, 84–90.

Washizaki, H., Yamamoto, H., and Fukazawa, Y. 2003. A metrics suite for measuring reusability of software components. In *9th IEEE International Software Metrics Symposium (METRICS'03)*. IEEE Computer Society, Sydney, Australia, 211–.

Watson, A. H. and McCabe, T. J. 1996. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, National Institute of Standards and Technology. August.

Weyuker, E. J. 1988. Evaluating software complexity measures. *IEEE Trans. Softw. Eng. 14,* 9, 1357–1365.

Yeh, P.-L. and Lin, J.-C. 1998. Software testability measurements derived from data flow analysis. In *2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, Florence, Italy, 96–103.

Yu, P., Systä, T., and Müller, H. A. 2002. Predicting fault-proneness using oo metrics: An industrial case study. In *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*. IEEE Computer Society, Budapest, Hungary, 99–107.

Zhou, Y. and Leung, H. 2006. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Software Eng. 32,* 10, 771–789.