Predicting Class Testability using Object-Oriented Metrics

Magiel Bruntink
CWI, P.O Box 94079
1098 SJ Amsterdam, The Netherlands
Magiel.Bruntink@cwi.nl

Arie van Deursen
CWI and Delft University of Technology
P.O Box 94079, 1098 SJ Amsterdam, The Netherlands
Arie.van.Deursen@cwi.nl

Abstract

In this paper we investigate factors of the testability of object-oriented software systems. The starting point is given by a study of the literature to obtain both an initial model of testability and existing OO metrics related to testability. Subsequently, these metrics are evaluated by means of two case studies of large Java systems for which JUnit test cases exist. The goal of this paper is to define and evaluate a set of metrics that can be used to assess the testability of the classes of a Java system.

1. Introduction

What is it that makes code hard to test? Why is one class easier to test than another? How can I tell that I'm writing a class that will be hard to test? What contributes to a class' *testability*? How can we quantify this notion?

Software testability is affected by many different factors, including the required validity, the process and tools used, the representation of the requirements, and so on — in the next section we will survey what has been written on this topic so far. This paper investigates testability from the perspective of unit testing, where our units consist of the classes of an object-oriented software system.

Our approach is to evaluate a set of object-oriented metrics with respect to their capabilities to predict the effort needed for testing. We choose this approach because metrics are a good driver for the investigation of aspects of software. The evaluation of metrics that are thought to have a bearing on the testing effort allows us, on the one hand, to gain insight into the factors of testability, and to obtain refined metrics on the other. Both results can subsequently be used in further studies.

The current popularity of the JUnit[1] framework for writing Java unit tests [3] provides a unique opportunity to pursue this approach and increase our understanding of testability. Test cases in JUnit are written in Java. A typical usage of JUnit is to test each Java class C by means of a dedicated test class C_T. Each pair $\langle C, C_T \rangle$ allows us to compare properties of C's source code with properties of its test class C_T. The route we pursue in this paper is to use these pairs to find source code metrics on C that are good predictors of test-related metrics on C_T.

We were triggered to investigate testability for two reasons. The first one is essentially scientific curiosity: while writing our own test cases we wondered why it was that for one class we had to think very hard before we were able to write a meaningful unit test suite, whereas for other classes we could generate test cases in a straightforward way. Thus, we started our research in order to get a better understanding of what contributes to good testability. Since testability holds a prominent place as part of the *maintainability* characteristic of the ISO 9126 quality model [13], this also increases our understanding of software quality in general.

Our second reason is more practical in nature: having quantitative data on testability is of immediate use in the software development process. The software manager can use such data to plan and monitor testing activities. The tester can use testability information to determine on what code to focus during testing. And finally, the software developer can use testability metrics to review his code, trying to find refactorings that would improve the testability of the code.

In this paper we describe our current results in characterizing software testability for object-oriented systems using source code metrics. We start out by surveying related work on (prediction of) testability. Then, in Section 3, we discuss factors affecting testability, narrowing down this very general notion to the source code perspective. In Section 4, we list our selection of object-oriented metrics, describe the experimental design, define metrics on JUnit test classes, and discuss the statistical techniques we use to investigate the relationships between these metrics. In Section 5 we describe the results of applying the experiment to Java systems accompanied by JUnit test cases. Moreover, we provide an in-depth discussion of the various (testability) factors that help to explain our results. We conclude by summarizing our contributions and listing areas of future work.

[1] Web: http://www.junit.org

2. Related Work

A number of testability theories have been published in the literature.

Voas et. al. [19] define software testability as the probability that a piece of software will fail on its next execution during testing, provided it contains a fault. This *fault sensitivity* is obtained by multiplying the probabilities that (1) the location containing the fault is executed; (2) the fault corrupts the program's state; and (3) the corrupted state gets propagated to the output. High fault sensitivity indicates high testability and vice versa.

Voas and Miller [20] present a different approach to fault sensitivity, in which semantic information contained in program specification and design documents is analyzed. An upper-bound on a component's fault sensitivity is given by the amount of information loss occurring within the component. Information loss can appear in roughly two guises: *Explicit* information loss occurs because the values of variables local to the component may not be visible at the system level, and thus cannot be inspected during testing. *Implicit* information loss is a consequence of the *domain/range ratio* (DRR) of the component. The DRR of a component is given by the ratio of the cardinality of the input to the cardinality of the output.

McGregor et. al. [15] attempt to determine the testability of an object-oriented system. They introduce the "visibility component" measure (VC for short), which can be regarded as an adapted version of the DRR measure. The VC has been designed to be sensitive to object oriented features such as inheritance, encapsulation, collaboration and exceptions. Furthermore, a major goal of the VC is the capability to use it during the early phases of a development process. Calculation of the VC will thus require accurate and complete specification documents.

Freedman [11] proposes "domain testability", based on the notions of observability and controllability as adopted in hardware testing. Observability captures the degree to which a component can be observed to generate the correct output for a given input. The notion of 'controllability' relates to the possibility of a component generating all values of its specified output domain. Adapting (the specification of) a component such that it becomes observable and controllable can be done by introducing extensions. Observable extensions add inputs to account for previously implicit states in the component. Controllable extensions modify the output domain such that all specified output values can be generated. Freedman proposes to measure the number of bits required to implement observable and controllable extensions to obtain an index of observability and controllability, and consequently a measure of testability.

Jungmayr [14] takes an integration testing point of view, and focuses on dependencies between components. He proposes the notion of *test-critical dependencies* as well as metrics to identify them and subsequently removing them using dedicated refactorings.

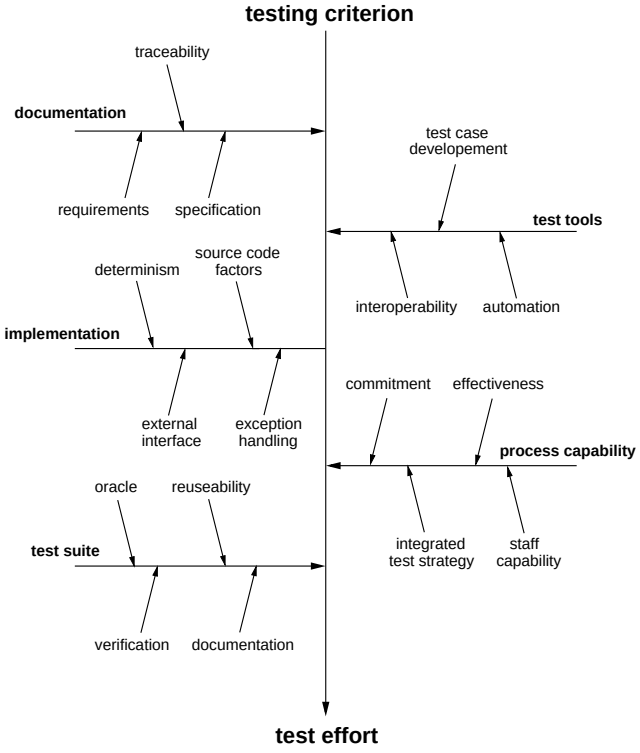

Figure 1. The testability fish-bone.

Focusing on testability analysis of UML class diagrams, Baudry et. al. [1] propose the use of various coupling and class interaction metrics to characterize testability.

Because of our focus on object-oriented white box unit testing we could not immediately reuse these models in our setting: some of the metrics contained in these models are not computable from source code alone; and not all assumptions (such as the assumption that testing proceeds in a black box fashion) applied to our setting. The model we use instead, which is partly based work by Binder [4], is discussed in Section 3.

3. Testability

The ISO defines testability as "attributes of software that bear on the effort needed to validate the software product" [13]. Binder [4] offers an analysis of the various factors that contribute to a system's testability. We will use an adapted version of his "fish bone" diagram (see Figure 1) to identify which factors of testability we address in the current paper.

3.1. Major Factors

Testing Criterion A major factor of the test effort picture is the degree of validity that the software is required to have. Based on the validity requirements, a project uses a *testing criterion* (or code coverage criterion) that specifies which parts of the software have to be tested. In effect, a testing criterion will establish a lower bound on the validity of the software, and an upper bound on the number of test cases required. A project will thus have to make a trade-off between the verification of validity on the one hand, and the required amount of testing on the other.

Documentation There are many reasons why a software system should be accompanied by documentation of several kinds. With regards to testing, requirements and specifications are of prime importance, capturing required and specified behavior, respectively. These documents should be correct and complete, and clear links should exist between the concepts captured in the various documents.

Implementation The implementation is the target of all testing, and thus the extent to which the implementation allows itself to be tested is a key factor of the testing effort. One important aspect of the implementation is determinism; it should be possible to consistently repeat tests.

The major part of the implementation of an application consists of source code expressed in one or more programming languages. Factors of the source code that relate to the testability of the implementation, and thus the testing effort, are the topic of this paper.

Test Suite Factors of the test suite itself also determine the effort required to test. Desirable features of test suites are correctness, automated execution and reuse of test cases. Similar to the system-under-test, test suites need documentation detailing the implemented tests, a test plan, test results of previous test runs and reports.

Test Tools The presence of appropriate test tools can alleviate many problems that originate in other parts of the 'fish bone' figure. For example, easy-to-use tools will demand less of the staff responsible for testing. Test case definition in the presence of graphical user interfaces is another example where tooling can significantly reduce the required effort.

Process Capability The organizational structure, staff and resources supporting a certain activity are typically referred to collectively as a (business) process. Properties of the testing process obviously have great influence on the effort required to perform testing. Important factors include a commitment of the larger organization to support testing, through funding, empowerment of those responsible, and provision of capable staff.

3.2. Source Code Factors

As a preparation for the experiments we describe in Section 4, we provide a short discussion of source code factors in general. Since our approach is exploratory in nature, rather than theoretical, the resulting 'model' of source code factors and their relation to testability is not intended to be complete.

We distinguish between two categories of source code factors: factors that influence the *number of test cases required* to test the system, and factors that influence the effort required to develop *each individual test case*. We will refer to the former category as test case generation factors, and to the latter category as test case construction factors.

3.2.1. Test Case Generation Factors

The number of test cases to be created and executed is determined by source code factors as well as the testing criterion. In many cases, the testing criterion determines which source code factors actually influence the number of required test cases. For example, McCabe's structured testing [21] generates test cases based on a program's control-flow. According to the structured testing criterion, a "basis set of paths through the control-flow graph of each module" has to be tested.

Object-oriented languages have some specific features that set them apart from procedural languages. These features may have their bearing on testing object-oriented programs. First, *inheritance* is a mechanism which allows classes to share their methods and fields. The set of methods and fields of a class is thus the union of the methods and fields that the class inherits, and those defined by the class itself. Depending on the object-oriented language, classes can also redefine the methods they inherit.

It is easy to see that inheritance is possibly a test case generation factor. For example, given that the project has decided to test all – inherited and defined – methods of each class, clearly the number of inherited methods of a class will influence the number of required test cases.

Second, *polymorphism* is a feature of object-oriented languages that allows objects to belong to multiple classes. Consider two classes, A and B. Let's say that A is the superclass of B. Now, objects of type class B are also objects of type class A. In practice this means that objects of type class B will be able to fill the role of objects of type class A. Because of this phenomenon, the use of polymorphic types can possibly increase the number of test cases required to cover all the cases. Of course, the testing criterion determines whether the additional cases have to be covered at all. For an up-to-date discussion of testing approaches for polymorphic types we refer the reader to [17].

3.2.2. Test Case Construction Factors

Once you know what needs to be tested according to your testing criterion, it may seem that creating the required test cases is a trivial task. However, as it turns out, the construction of test cases is at least as difficult a problem as finding

out what you need to test. For example, McCabe's structured testing criterion can require that unreachable control-flow paths are exercised.

Even if a test case is constructible in theory, there are source code factors that influence the effort needed to construct it. The class-under-test will need to be initialized such that effective testing can be done. In our case, this entails that the fields of (an object of) a class are set to the right values before a test case can be executed. Furthermore, if the class-under-test depends on other classes – because it used members of those classes – those will need to be initialized. A class which deals with external interfaces (hardware, etc.) will typically require the external components to be initialized as well. Due to our focus on source code factors, we will not consider the latter source of initialization work. In sections 4 and 5 we investigate whether or not these source code factors influence the testing effort.

4. Experimental Design

The objective of this paper is to evaluate to what extent a number of well-known object-oriented metrics can be used to predict class testability. In this section, we first (4.1) identify possibly relevant metrics, and then we turn this relevance into a hypothesis to be analyzed (4.2). Subsequently, we describe test suite metrics (4.3), as well as the statistical methods that can be used to evaluate the hypothesis (4.4).

4.1. Object-Oriented Metrics

What metrics are suitable candidates for characterizing the testability factors discussed in the previous section? We will use a metrics suite proposed by Binder [4] as starting point. Binder is interested in testability as well, and uses a model distinguishing "complexity" and "scope" factors, which are similar to our test case construction and generation factors. Unfortunately, Binder does not provide an operational definition of the metrics used. In this section, we define each metric used operationally.

The metrics used by Binder are based on the well known metrics suite provided by Chidamber and Kemerer [9], who for some of their metrics (such as the Coupling Between Objects and the Response for Class) already suggested that they would have a bearing on test effort.

In the related work section (2) we discuss various alternative metrics indicative of testability. The metrics discussed below and used in our experiment have the advantage of being easier to implement and understand.

4.1.1. Notation

To give each metric a concise and unambiguous definition, we use the notation introduced by Briand et. al. [7].

Definition 1 (Classes) *An object-oriented system consists of a set of classes, C. For every class $c \in C$ we have:*

- *$Parent(c) \subset C$, the parent class of c.*
- *$Children(c) \subset C$, the set of classes that inherit directly from c.*
- *$Ancestors(c) \subset C$, the set of classes from which c inherits either directly or indirectly.*

Definition 2 (Methods) *Let $c \in C$, then we have:*

- *$M_{In}(c)$, the set of methods that c inherits.*
- *$M_D(c)$, the set of methods that c newly declares, i.e. $m \in M_D(c)$ iff m is declared in c and $m \notin M_{In}$.*
- *$M_{Im}(c)$, the set of methods that c implements, i.e. c defines a body for m. Clearly, for each $m \in M_{Im}(c)$ either $m \in M_{In}$ or $m \in M_D$.*
- *$M(c) = M_D(c) \cup M_{In}(c)$, the set of methods of c.*
- *$M(C) = \cup_{c \in C} M(c)$, the set of all methods.*

Definition 3 (Method Invocations) *Let $c \in C$, $m \in M_{Im}(c)$ and $m' \in M(C)$, then we have:*

- *$MI(m)$, the set of methods invoked by m. $m' \in MI(m)$ iff the body of m contains an invocation of method m'.*

Definition 4 (Fields) *Let $c \in C$, then we have:*

- *$F_{In}(c)$, the set of fields that c inherits.*
- *$F_D(c)$, the set of fields that c newly declares, i.e. $f \in F_D(c)$ iff f is declared in c and $f \notin F_{In}$.*
- *$F(c) = F_{In} \cup F_D$, the set of fields of c.*
- *$F(C) = \cup_{c \in C} F(c)$, the set of all fields.*

Definition 5 (Field References) *Let $m \in M_{Im}(c)$ for some $c \in C$ and $f \in F(C)$, then we have:*

- *$f \in FR(m)$ iff the body of m contains a reference to f.*

4.1.2. Metrics

We now introduce the metrics that we evaluate in our experiments. Let $c \in C$ be a class.

Depth of Inheritance Tree

$$DIT(c) = |Ancestors(c)|$$

The definition of DIT relies on the assumption that we deal with object-oriented programming languages that allow each class to have at most one parent class; only then will the number of ancestors of c correspond to the depth of c in the inheritance tree. Our subject language – Java – complies to this requirement, however C++ does not.

Fan Out

$$FOUT(c) = |\{d \in C - \{c\} : uses(c,d)\}|$$

where $uses(c,d)$ is a predicate that is defined by:

$$\begin{aligned} & uses(c,d) \leftrightarrow \\ & \quad (\exists m \in M_{Im}(c) : \exists m' \in M_{Im}(d) : m' \in MI(m)) \\ \vee & \quad (\exists m \in M_{Im}(c) : \exists a \in F(d) : a \in FR(m)) \end{aligned}$$

In words, $uses(c,d)$ holds if and only if a method of c either calls a method or references a field of d.

Lack Of Cohesion Of Methods

$$LCOM(c) = \frac{\left(\frac{1}{a}\sum_{f \in F_D(c)} \mu(f)\right) - n}{1-n}$$

where $a = |F_D(c)|$, $n = |M_{Im}(c)|$ and $\mu(g) = |\{m \in M_{Im}(c) : g \in FR(m)\}|$, the number of implemented methods of class c that reference field g.

This definition of LCOM is proposed by Henderson-Sellers in [12]. It is easier to both compute and interpret compared to Chidamber and Kemerer's definition of the metric. The metric yields 0, indicating perfect cohesion, if all the fields of c are accessed by all the methods of c. Conversely, complete lack of cohesion is indicated by a value of 1, which occurs if each field of c is accessed by exactly 1 method of c. It is assumed that each field is accessed by at least one method, furthermore, classes with one method or no fields pose a problem; such classes are ignored during calculation of the LCOM metric.

Lines Of Code Per Class

$$LOCC(c) = \sum_{m \in M_{Im}(c)} LOC(m)$$

where LOC(m) is the number of lines of code of method m, ignoring both blank lines and lines containing only comments.

Number Of Children

$$NOC(c) = |Children(c)|$$

Number Of Fields

$$NOF(c) = |F_D(c)|$$

Number Of Methods

$$NOM(c) = |M_D(c)|$$

Response For Class

$$RFC(c) = |M(c) \cup_{m \in M(c)} MI(m)|$$

The RFC of c is a count of the number of methods of c and the number of methods of other classes that are invoked by the methods of c.

Weighted Methods Per Class

$$WMC(c) = \sum_{m \in M_{Im}(c)} \mathrm{VG}(m)$$

where VG(m) is McCabe's cyclomatic complexity number[21] for method m.

4.2. Goal and Hypotheses

We set up our experiments for evaluating this set of metrics using the GQM/MEDEA[2] framework proposed by Basili et. al. [8]. First, we describe the *goal*, *perspective* and *environment* of our experiment:

Goal: To assess the capability of the proposed object-oriented metrics to predict the testing effort.

Perspective: We evaluate the object-oriented metrics at the class level, and limit the testing effort to the unit testing of classes. Thus, we are assessing whether or not the values of the object-oriented metrics can predict the required amount of effort needed for unit testing a class.

Environment: The experiments are targeted at Java systems, which are unit tested at the class level using the JUnit testing framework. Further relevant factors of these systems will be described in Section 5.

The JUnit framework allows the user to create (and run) classes that are capable of unit testing a part of the system. A typical practice is to create a test class for every class of the system. We target at subject systems in which there is one test class responsible for the unit testing of each class.

To help us translate the goal into measurements, we pose questions that pertain to the goal:

Question 1: Are the values of the object-oriented metrics for a class correlated to the required testing effort for that class?

Answering this question directly relates to reaching the goal of the experiments. However, to answer it, we must first quantify "testing effort." To indicate the testing effort required for a class we use the size of the corresponding

[2]Goal Question Metric / MEtric DEfi nition Approach

test suite. Well-known cost models such as Boehm's COCOMO [6] and Putnam's SLIM model [16] relate development cost and effort to software size. Test suites are software in their own right; they have to be developed and maintained just like 'normal' software. In Section 4.3 we will define precisely which metrics we use to measure the size of a test suite.

Now we refine our original question, and obtain the following new question:

Question 2: Are the values of the object-oriented metrics for a class correlated to the size of the corresponding test suite?

From these questions we derive the hypotheses that our experiments will test:

$H_0(m,n)$: There is *no* correlation between object-oriented metric *m* and test suite metric *n*,

$H_1(m,n)$: There is a correlation between object-oriented metric *m* and test suite metric *n*,

where *m* ranges over our set of object-oriented metrics, and *n* over our set of test-suite based metrics.

4.3. Test Suite Metrics

For our experiments we propose the dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases) metrics to indicate the size of a test suite. The 'd' prepended to the names of these metrics denotes that they are the *dependent* variables of our experiment, i.e. the variables we want to predict. The dLOCC metric is defined like the LOCC metric.

The dNOTC metric provides a different perspective on the size of a test suite. It is calculated by counting the number of invocations of JUnit 'assert' methods that occur in the code of a test class. JUnit provides the tester with a number of different 'assert' methods, for example 'assertTrue', 'assertFalse' or 'assertEqual'. The operation of these methods is the same; the parameters passed to the method are tested for compliance to some condition, depending on the specific variant. For example, 'assertTrue' tests whether or not its parameter evaluates to 'true'. If the parameters do not satisfy the condition, the framework generates an exception that indicates a test has failed. Thus, the tester uses the set of JUnit 'assert' methods to compare the expected behavior of the class-under-test to its current behavior. Counting the number of invocations of 'assert' methods, gives the number of comparisons between expected and current behavior which we consider an appropriate definition of a test case.

4.4. Statistical Analysis

In order to investigate our hypotheses, we calculate Spearman's rank-order correlation coefficient, r_s, for each object-oriented metric of the system classes and both the dLOCC and dNOTC metrics of the corresponding test classes. We use $r_s(m,n)$ to denote Spearman's rank-order correlation between object-oriented metric *m* and test suite metric *n*.

Spearman's rank-order correlation coefficient is a measure of association between two variables that are measured in at least an ordinal scale [18]. The measurements are *ranked* according to both variables. Subsequently, the measure of association is derived from the level of agreement of the two rankings on the rank of each measurement. We decided to use this correlation measurement (and not the more common Pearson correlation), since r_s can be applied independent of the underlying data distribution, and independent of the nature of the relationship (which need not be linear).

In order to test our hypothesis we estimate the statistical significance of the observed value of r_s by calculating the *t* statistic [18]. This statistic provides a significance *p* (which indicates the probability that the observed value is a chance event) based on the number of pairs in the data set. The signficance *p* allows us to reject $H_0(m,n)$ (and accept $H_1(m,n)$) with a certain level of confidence.

To calculate r_s, we need to find the corresponding test class for every system class. The JUnit documentation suggests that test classes should be named after the class they test, by appending "Test" to the class' name, a convention used in both our subject systems.

Both r_s and *t* are calculated for each object-oriented metric *m* and the dLOCC and dNOTC metrics of the test suite. First, the values for all classes for object-oriented metric *m* are fetched from a repository. Subsequently, each value for a class is paired with both the dLOCC and dNOTC values of the corresponding test class. The resulting pairs are then used to calculate r_s. Finally, *t* is derived from the value of r_s and the number of pairs involved, and the statistical significance (*p*) of *t* is obtained from a standard table [18]. This process is repeated for all the object-oriented metrics in our set, and finally the results are presented in a table (see section 5).

5. Case Studies

We used two software systems for our experiments. The first is DocGen, a commercial source code documentation tool in development at the Software Improvement Group. The second is Apache Ant, an open source automation tool for software development. Both systems are unit tested at the class level by means of JUnit. We first present the results of the experiments, and then discuss them at the end of the section.

5.1. DocGen

DocGen is a documentation generator, developed by the Software Improvement Group (SIG). It processes source code of other programs, and generates technical documen-

tation based on facts that are contained in the source code. DocGen is described in more detail in [10].

The DocGen source code consists of 90,000 lines of Java code, divided over 66 packages containing 640 classes in total. Of these 640 classes, 138 classes have an associated test class. The classes which have an associated test class are used for our experiment.

The development of DocGen is based on Kent Beck's eXtreme Programming (XP) methodology [2]. The use of *coding standards* has a clear implication for the structure of the code; methods are not allowed to exceed 12 lines. Additionally, the testing of DocGen is not a separate process; it is an integral part of the development process. The DocGen developers take testing very seriously, and they are convinced that they do a good job on software testing. There is, however, no explicit testing criterion in use: programmers select test cases based on personal judgement.

5.2. Apache Ant

Ant[3] is a build tool, and is being developed as a subproject of the Apache web server. Ant's source code is larger than Docgen, and comprises 170,000 lines of Java code. There are 887 classes contained in 87 packages. Again, our experiment uses the classes that have an associated test class; there are 111 such classes. The Ant source code is kept in a public CVS repository, from which we obtained the 1.5.3 branch, dated April 16, 2003.

The testing process at the Ant project is similar to that of DocGen. Programmers develop JUnit test cases during development, and run these tests nightly. Additionally, the functional correctness of the entire system is verified every night by running Ant in a typical production environment. Again, there is no explicit testing criterion; test cases are created based on the preference of the programmers.

5.3. Data Collection

In order to collect data from subject systems, we have used the Eclipse platform[4] to calculate the metrics. An existing plug-in for Eclipse, the "Eclipse metrics plug-in"[5], has been extended to calculate our set of metrics for a given system, and to store the results in a source code repository based on a relational database.

5.4. Results

Table 1 holds the results of the experiments described in Section 4 for both DocGen and Ant. The table contains the values of Spearman's rank-order correlation coefficient (r_s) for object-oriented metric *m* and both test suite metrics dLOCC and dNOTC. The object-oriented metrics which are significantly correlated to the test suite metrics at the 99% confidence level, are set in boldface in Table 1. For these, we can reject $H_0(m,n)$ and accept $H_1(m,n)$. Note that the accepted hypotheses for Ant and DocGen are the same, except for the LCOM metric, which is significantly correlated in Ant, but not in DocGen.

DocGen	dLOCC	dNOTC	Ant	dLOCC	dNOTC
DIT	-.03675	-.0590	DIT	-.0456	-.201
FOUT	.5549	.457	**FOUT**	.465	.307
LCOM	.166	.207	**LCOM**	.437	.382
LOCC	.513	.518	**LOCC**	.500	.325
NOC	-.0274	.00241	NOC	.0537	-.0262
NOF	.248	.233	**NOF**	.455	.294
NOM	.355	.401	**NOM**	.532	.369
RFC	.537	.520	**RFC**	.526	.341
WMC	.422	.460	**WMC**	.531	.348

Table 1. Spearman's r_s values.

Additionally, Table 2 provides the correlations among the object-oriented metrics themselves.

5.5. Discussion

What can we learn from the data collected from Table 1? How can we explain these figures? What do these figures tell us about the impact of source code factors and testability? In this section we will discuss these results, making use of the test case generation (number of test cases) and test case construction (complexity of the test cases) factors as discussed in Section 3.

A first observation to make is that the source code metrics themselves are correlated: we naturally expect that a large class (high LOCC) has a large number of methods (high NOM). The correlations between the various metrics are listed in Table 2. We use these correlations to organize our discussion, and cover clusters of correlated metrics.

A second observation is that the test suite metrics are correlated as well: the larger a test class (dLOCC), the more assertions it will contain (dNOTC).

However, some source code metrics are better predictors of dLOCC than of dNOTC. Using Hotelling's *t* test to determine the statistical significance of the difference between two correlations, we find that for DocGen, fan out (FOUT) is a significantly better predictor of the number of lines of test class (dLOCC) than the number of test cases (dNOTC). For Ant, the metrics FOUT, LOCC, RFC, NOF, NOM and WMC are significantly better predictors of dLOCC than of dNOTC.

Apparently these metrics predict a factor of test classes which distinguishes dLOCC and dNOTC. We conjecture that a distinguishing factor might be the effort, expressed in lines of code, required to construct the test cases, i.e. describe the test cases themselves, and provide sufficient initialization of the system. In effect, the object-oriented metrics which

[3]Web: `http://ant.apache.org`
[4]Web: `http://www.eclipse.org`
[5]Web: `http://sourceforge.net/projects/metrics`

DocGen	DIT	FOUT	LCOM	LOCC	NOC	NOF	NOM	RFC	WMC
DIT	1								
FOUT	-.0287	1							
LCOM	-.0879	.317	1						
LOCC	-.129	.691	.379	1					
NOC	.0511	.0506	.0134	.204	1				
NOF	-.252	.365	.766	.444	.0520	1			
NOM	.0750	.546	.481	.847	.226	.443	1		
RFC	-.00672	.837	.420	.894	.184	.432	.867	1	
WMC	-.0351	.579	.436	.931	.208	.421	.952	.879	1

Ant	DIT	FOUT	LCOM	LOCC	NOC	NOF	NOM	RFC	WMC
DIT	1								
FOUT	.120	1							
LCOM	.0201	.306	1						
LOCC	.142	.911	.311	1					
NOC	-.0762	-.0794	-.0723	.0289	1				
NOF	.105	.686	.462	.747	.109	1			
NOM	.00869	.704	.436	.819	.216	.825	1		
RFC	.150	.929	.344	.944	.0229	.789	.861	1	
WMC	.126	.865	.354	.975	.0963	.784	.899	.945	1

Table 2. r_s values between the OO metrics.

predict dLOCC better than dNOTC would then measure test case construction factors. Below we provide more discussion on test case construction factors for the individual metrics.

5.5.1. Size-Related Metrics

The first cluster of metrics we discuss measures the size of the source code. These include the lines of code (LOCC), and the number of fields (NOF) and methods (NOM and WMC).

Naturally, we expect that a large class needs a large corresponding test class, so we are not surprised to see that all four metrics are correlated with both test suite metrics in Ant as well as DocGen. The size of the class is first of all a test case generation factor: a larger class requires more test cases (higher dNOTC). At the same time, a larger class may be harder to test (higher dLOCC), because of intra-class dependencies, making size a test case construction factor as well.

Number Of Fields (NOF)

The fields of the class-under-test need to be initialized before testing can be done. We argued before that the amount of required initialization influences the testing effort and the dLOCC metric. Thus, we expect correlation between the NOF and dLOCC metrics. However, for DocGen the correlation we observe is only weak (but significant), while for Ant it is moderate. Neither is the correlation between NOF and dLOCC significantly better than the correlation between NOF and dNOTC for DocGen, though it is for Ant. A possible explanation is given by the definition of the NOF metric. In section 4 NOF(c) is defined by NOF(c) $= |F_D(c)|$. In words, NOF(c) is a count of the number of fields class c (newly) declares. The number of fields that class c inherits from its ancestors is therefore not included in the count. If classes tend to use fields they have inherited, the NOF metric may not be a sufficient predictor of the initialization required for testing. Whether or not this explains the difference between the observed correlations for DocGen and Ant remains the subject of further research.

Number of Methods (NOM)

One would expect that the number of methods primarily affects the number of test cases to be written (at least one test case per method), and not the complexity required to write the test cases. Thus, NOM is a test case generation rather than construction factor, and we would expect NOM to predict dLOCC and dNOTC equally well. DocGen indeed lives up to the expectation.

However, in the case of Ant, the difference between the dLOCC and dNOTC correlations is significant. A possible explanation for this is that for Ant the correlation between the NOM and NOF metrics is strong (see Table 2), i.e. the number of methods of a class is a strong predictor of the number of fields of a class. We saw before how the number of fields of a class can influence the effort needed to test, i.e. the dLOCC metric. Thus, the correlation between the NOM and dLOCC metrics for Ant could be explained indirectly via the NOF metric. The fact that the correlation between the NOM and NOF metrics is only moderate for DocGen confirms this explanation.

Weighted Methods Per Class (WMC)

The WMC metric also counts the methods per class, but weighs them with McCabe's cyclomatic complexity number.

We observe that the WMC metric correlates strongly with the NOM metric for both DocGen and Ant (see Table 2). Also, the relationships to the other metrics are very similar for both WMC and NOM. An explanation is offered by the fact that for both systems, the VG value of each method tends to be low, and close to the average. For DocGen, we have an average VG of 1.31, with standard deviation of 0.874 and maximum of 17. For Ant, we have an average VG of 2.14, with standard deviation of 2.91 and maximum of 61. Thus, for our systems the WMC metric will tend to measure the number of methods, i.e. the NOM metric. We conclude that the same effects explain the correlations with the test suite metrics for both WMC and NOM.

As a side note, the low average, standard deviation and maximum values of the VG of the methods of DocGen are a result of a coding standard in use at the SIG. According to the coding standard, each method should not contain more than 12 lines of code.

5.5.2. Inheritance-Related Metrics

The second group of metrics we consider deals with inheritance: DIT measures the superclasses (the depth of the inheritance tree), whereas NOC measures the subclasses (the number of children). Somewhat surprisingly, neither of these

metrics are correlated to any test suite metrics in either case study.

Under what test strategies would these metrics be good indicators for the size of a test class? For DIT, if we require that all inherited methods are retested in any subtype (see, e.g., [5]), the depth of inheritance DIT metric is likely to be correlated with test suite size — assuming that more superclasses lead to more inherited fields and methods.

For NOC, we would obtain a correlation with test size if our test strategy would prescribe that classes that have many subtypes are more thoroughly tested. This would make sense, since errors in the superclass are likely to recur in any subclass. Moreover, if the classes are designed properly (adhering to the Liskov Substitution Principle), superclass test cases can be reused in any subclass (see, e.g., [5]).

From the fact that DIT nor NOC are correlated with test class size, we conclude that in the subject systems studied these strategies were not adopted by the developers.

5.5.3. External Dependencies

The Fan Out (FOUT) and Response-For-Class (RFC) metrics measure dependencies on external classes (FOUT) and methods (RFC). In both case studies these metrics are significantly correlated with both test suite metrics.

Fan Out (FOUT)
An interesting property of FOUT is that it is a significantly better predictor of the dLOCC metric than of the dNOTC metric (at the 95% confidence level for DocGen, 99% for Ant). The fan out of a class measures the number of other class es that the class depends on. In the actual program, (objects of) these classes will have been initialized before they are used. In other words, the fields of the classes will have been set to the appropriate values before they are used. When a class needs to be (unit) tested, however, the tester will need to take care of the initialization of the (objects of) other classes and the class-under-test itself. The amount of initialization required before testing can be done will thus influence the testing effort, and by assumption, the dLOCC metric. By this argument, the FOUT metric measures a test case construction factor.

Response For Class (RFC)
From the definition in Section 4, it is clear that the RFC metric consists of two components. First, the number of methods of class *c*. The strong correlation between the RFC and NOM metrics for both systems is explained by this component. Second, the number of methods of other classes that are potentially invoked by the methods of *c*. The invocation of methods of other classes gives rise to fan out, hence the strong correlation between RFC and FOUT in both systems. Given the correlations between the RFC metric and both the NOM and FOUT metrics, the observed correlations between the RFC and dLOCC metrics for both DocGen and Ant are as expected.

5.5.4. Class Quality Metrics

The last metric to discuss is the Lack of Cohesion of Methods (LCOM). This metric is interesting, in the sense that it is signficantly correlated with test suite metrics for Ant, but not for DocGen.

To understand why this is the case, observe that Table 2 shows that for both systems the LCOM and NOF metrics are moderately correlated. In case of DocGen, the correlation is even fairly strong. Thus, it seems that for our case studies, classes that are not cohesive (high LCOM value) tend to have a high number of fields, and similarly, classes that are cohesive tend to have a low number of fields. Similar correlations exist between the LCOM and NOM metrics. Thus, incohesive classes tend to have a high number of fields and methods, and cohesive classes tend to have a low number of fields and methods. These effects are intuitively sound: it is harder to create a large cohesive class than it is to create a small one.

6. Concluding Remarks

The purpose of this paper is to increase our understanding of what makes code hard to test. To that end, we analyzed relations between classes and their JUnit test cases in two Java systems totalling over 250,000 lines of code. We were able to demonstrate a significant correlation between class level metrics (most notably FOUT, LOCC, and RFC) and test level metrics (dLOCC and dNOTC). Moreover we discussed in detail how various metrics can contribute to testability, using an open source and a commercial Java system as example systems.

Our approach is based on an extensive survey of the literature on software testability. During our survey, we were not able to find other papers analyzing relationships between source code and test data. We conducted our experiments using the GQM/MEDEA framework, and we evaluated our results using Spearman's rank-order correlation coefficient. Finally, we offered a discussion of factors that can explain our findings.

We consider our results a necessary and valuable first step for understanding what makes code harder to test. We foresee the following extensions of our work.

First, our experimental basis should be extended. Although the systems studied were large, containing many ⟨class, testclass⟩ pairs, it is desirable to extend our findings to a larger number of systems, developed by all sorts of teams using different development methodologies. We are in the

process of making our toolset available so that others can repeat our experiments on their own systems.

Second, we see opportunities for enriching the underlying testability model. At present we focus on unit level ⟨class, testclass⟩ pairs. It would be interesting to see how package level unit testing would fit in, or perhaps package level functional testing. As an example, the latter approach is used in the Eclipse implementation, in which JUnit is used to implement (package level) functional tests.

Another possibility is the use of more powerful statistics than Spearman's rank-order correlation in order to further assess the predictive capabilities of the metrics. Recent work by Wheeldon and Counsell[22] shows that many object-oriented coupling metrics obey power law distributions. Since metrics like FOUT and RFC are related to coupling metrics, they may be distributed similarly. Consequently, more powerful statistics could possibly be used to assess them.

Last but not least, the metrics we propose can be incorporated in an IDE such as Eclipse, offering help to both the tester and the developer. Currently Eclipse is well-integrated with JUnit, offering, for example, a button to generate a JUnit xxTest class body for a given class xx. Our metrics can be used to support the test process, not only in order to identify classes that are hard to test, but also to signal ⟨class, testclass⟩ pairs that deviate from normal findings (or from metrics results obtained from a system that is considered thoroughly tested) allowing a tool to issue testability-related warnings.

Acknowledgments Thanks to Tobias Kuipers from the Software Improvement Group for his support throughout the project. Thanks to Adam Booij for his help with the calculation of the statistics, and his help with statistics in general. We thank Tom Tourwé, Tijs van der Storm, Jurgen Vinju and Vania Marangozova for commenting on drafts of this paper. Partial support was received from ITEA (Delft University of Technology, project MOOSE, ITEA 01002).

References

[1] B. Baudry, Y. Le Traon, and G. Sunyé. Testability analysis of a UML class diagram. In *Proceedings of the Ninth International Software Metrics Symposium (METRICS03)*, pages 54–66. IEEE Computer Society, 2003.

[2] K. Beck. *eXtreme Programming eXplained*. Addison-Wesley, Reading, Massachusetts, 1999.

[3] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[4] R. Binder. Design for testability in object-oriented systems. *Comm. of the ACM*, 37(9):87–101, 1994.

[5] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[6] B.W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[7] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Tr. on Software Engineering*, 25(1):91–121, 1999.

[8] L. C. Briand, S. Morasca, and V. Basili. An operational process for goal-driven definition of measures. *IEEE Transactions on Software Engineering*, 28(12):1106–1125, December 2002.

[9] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[10] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 40–49. IEEE Computer Society, 1999.

[11] R. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[12] B. Henderson-Sellers. *Object-Oriented Metrics*. Prentice Hall, New Jersey, 1996.

[13] ISO. International standard ISO/IEC 9126. information technology: Software product evaluation: Quality characteristics and guidelines for their use, 1991.

[14] S. Jungmayr. Identifying test-critical dependencies. In *Proceedings of the International Conference on Software Maintenance*, pages 404–413. IEEE Computer Society, October 2002.

[15] J. McGregor and S. Srinivas. A measure of testing effort. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 129–142. USENIX Association, June 1996.

[16] L.H. Putnam. A general empirical solution to the macrosoftware sizing and estimating problem. *IEEE Tr. on Software Engineering*, 4(4):345–61, 1978.

[17] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 2003.

[18] S. Siegel and N. J. Castellan Jr. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill Book Company, New York, 1988.

[19] J. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[20] J. Voas and K. Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20:207–216, March 1993.

[21] A. Watson and T. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. T. NIST Special Publication 500-235, National Institute of Standards and Technology, Washington, D.C., 1996.

[22] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, September 2003.