Available online at www.sciencedirect.com

SCIENCE DIRECT®

The Journal of Systems and Software 79 (2006) 1219–1232

The Journal of Systems and Software

www.elsevier.com/locate/jss

An empirical study into class testability ☆

Magiel Bruntink [a,*], Arie van Deursen [a,b]

[a] CWI, P.O. Box 94079, 1098 SJ Amsterdam, The Netherlands

[b] Delft University, Faculty of Electrical Engineering, Mathematics and Computer Science, Mekelweg 4, 2628 CD Delft, The Netherlands

Received 26 January 2006; accepted 20 February 2006
Available online 31 March 2006

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 object-oriented metrics related to testability. Subsequently, these metrics are evaluated by means of five case studies of commercial and open source Java systems for which JUnit test cases exist. The goal of this paper is to identify and evaluate a set of metrics that can be used to assess the testability of the classes of a Java system.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Testability; Software metrics; Object-oriented systems; Software testing; Unit testing

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 conduct an empirical study in order to evaluate a set of object-oriented source code metrics with respect to their capabilities to predict the effort needed for testing. 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 (Beck and Gamma, 1998) 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.

The subject systems of our empirical study are five Java systems for which JUnit test suites were available. The systems, which total over 290,000 lines of code, include four commercial systems and one open source system (Ant). All systems were developed using an agile process based mostly on extreme programming (Beck, 1999). Three of the systems were developed with an explicit testing criterion such as 90% branch coverage: the other two achieve 70–80% statement coverage, but this was not an explicitly stated criterion.

☆ This is a substantially revised version of our paper *Predicting Class Testability using Object-Oriented Metrics* that appeared in the Proceedings of Fourth International Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2004.

* Corresponding author. Tel.: +31 20 5924222; fax: +31 20 5924199.
E-mail addresses: Magiel.Bruntink@cwi.nl (M. Bruntink), Arie.van.Deursen@cwi.nl (A. van Deursen).

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

Our motivation to investigate testability metrics is two-fold. The first originates essentially from 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 ISO, 1991, 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 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 offer a survey of JUnit and the typical way it is used, which comprises the context of our experiment. In Section 5, 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 6 we summarize the key characteristics of the systems under study, while in Section 7 we survey the results of our experiments, and offer an in-depth discussion of the various (testability) factors that help to explain our results. In Section 8 we assess the validity of our results, after which we conclude by summarizing our contributions and listing areas of future work.

2. Related work

A number of testability theories have been published in the literature. Voas (1992) 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 (1995) 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 and Srinivas (1996) 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 (1991) 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. Observability 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.

Freedman's work on observability and controllability has been continued by Le Traon et al. who provide an axiomitization of key properties of testability, such as *monotonicity of design composition*: "a composite design is less testable than any of its components" (Traon et al., 2000). Further research in the same direction is by Nguyen et al. who generate testability models for embedded software by analyzing the corresponding *static single assignment* representation of that software (Nguyen et al., 2002; Nguyen et al., 2003).

Jungmayr (2002) 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.

Concerning testability analysis of UML class diagrams, Baudry et al. (2002, 2003) propose the use of various coupling and class interaction metrics to characterize testability. Their focus is on integration and subsystem testing. They define "testability anti-patterns", which capture situations in which class interactions are overly complex and increase the testing effort. This work was also presented at a recent workshop devoted entirely to analyzing testability of software systems (du Bousquet and Delaunay, 2004).

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 on work by Binder (1994), is discussed in Section 3.

Several metrics-based approaches for predicting quality attributes related to testability, such as reliability and maintainability have been published. Examples include maintainability predictions correlating metrics and estimated maintenance efforts (Dagpinar and Jahnke, 2003; Muthanna et al., 2000), the identification of fault-prone components (Khoshgoftaar, 2001; Nikora and Munson, 2003), and the defect detectors from Menzies et al. (2003). These approaches are similar to ours in the sense that they use statistical methods to predict external attributes such as mean time between failure or maintenance costs from internal attributes such as the source code or design artifacts.

3. Testability

The ISO defines testability as "attributes of software that bear on the effort needed to validate the software product" (ISO, 1991). Binder (1994) offers an analysis of the various factors that contribute to a system's testability, which he visualizes using the fish bone diagram as shown in

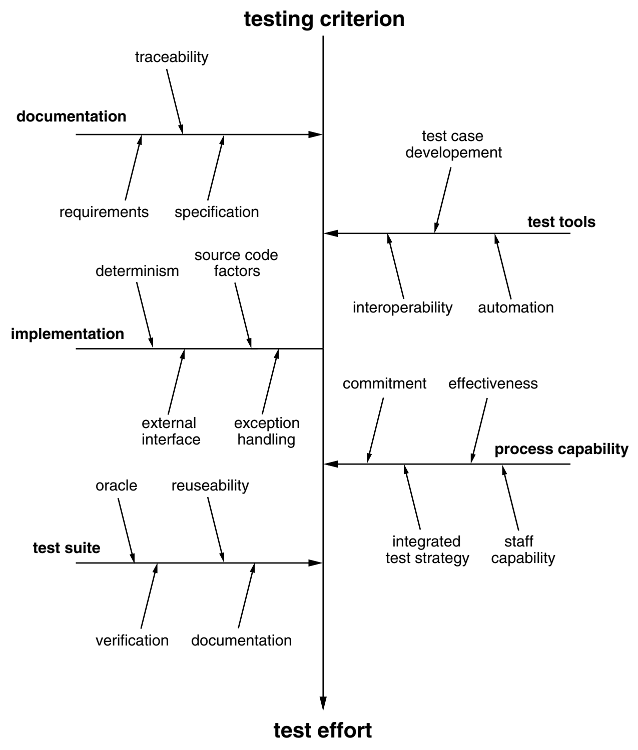

Fig. 1. The testability fish-bone.

Fig. 1. The major factors determining test effort Binder distinguishes include the testing criterion that is required, the usefulness of the documentation, the quality of the implementation, the reusability and structure of the test suite, the suitability of the test tools used, and the process capabilities.

Of these factors, our study is concerned with the structure of the implementation, and on source code factors in particular. 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, which both are discussed below.

3.1. 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 (Watson and McCabe, 1996) 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. A basis set of paths consists of linearly independent paths that combine to form the set of all paths.

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 the specific method body to be executed to be determined dynamically, at run time. The use of polymorphism can also result in a need for more test cases, and hence can be considered a test case generation factor. A typical example is the *use* of a polymorphic method m of superclass S at a given call site. A testing criterion such as *all-target-methods* requires that test cases are derived ensuring that each possible binding of m to a body from a subclass of S is exercised. An even stricter criterion is *all-receiver-classes*, which insists that the call to m is exercised for every possible subclass of S. For an up-to-date discussion of testing approaches for polymorphic types we refer the reader to Rountev et al. (2004).

3.2. Test case construction factors

Once we know what needs to be tested according to the 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 needs to be tested. 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 5 and 6 we investigate whether or not these source code factors influence the testing effort.

4. Class testing using the xUnit framework

In this section we give some background about the xUnit family of testing frameworks. Moreover, we describe how xUnit is typically used for class-level unit testing. As such, it offers the context of our experiments, and we expect our findings to hold for software that is tested under similar conditions.

The xUnit family of testing frameworks offers support for class-level unit testing (Beck, 1994; Beck and Gamma, 1998). Instantiations for numerous languages exist, including Smalltalk (sunit), all .NET languages (nunit), C++ (cppunit), and Java (JUnit), which is the one we will use in this paper. JUnit has become popular as part of the extreme programming and agile software development methodologies.[2] JUnit is used widely in industry as well as in software engineering and testing courses taught in academia.

Test cases in JUnit are written in Java. The typical use is to create for every class `Foo.java` a corresponding test class called `FooTest.java`. This test class extends the JUnit `TestCase` class, and thus can be activated by means of the JUnit test driver.

A test class can contain a number of test cases. These test cases may share the need to setup a common object structure (the *fixture*), which can be used to activate the method under test. This is facilitated by JUnit's `setUp` method, which is automatically invoked before each test case.

The test cases themselves are represented by a method having a name that starts with `test`. Each test typically refines the fixture so that the method under test can be

```
public void setUp() {
    project = new Project();
    project.setBasedir(".");
}
...
public void testNestedPatternset() {
    PatternSet p = new PatternSet();
    p.setIncludes("**/*.java");

    PatternSet nested = new PatternSet();
    nested.setExcludes("**/*.class");

    p.addConfiguredPatternset(nested);

    String[] excludes = p.getExcludePatterns(project);
    String[] includes = p.getIncludePatterns(project);

    assertEquals("Includes", "**/*.java", includes[0]);
    assertEquals("Excludes", "**/*.class", excludes[0]);
}
```

Fig. 2. Example JUnit test method for Ant pattern sets.

invoked. After carrying out the method under test, the resulting object structures are inspected, and compared to expected results. For this, a series of `assertEquals` methods is provided.

An example is shown in Fig. 2, which contains a test case from `Ant` – one of the systems we have used in our experiments (see Section 6.2). The example shows the `setUp` method which creates a new Ant `Project` needed by all test cases in the same class, as well as one test method, which creates a "nested pattern", and then inspects (by means of two `assertEquals` invokations) that the outer pattern set includes the inner pattern set. The three arguments of the `assertEquals` method represent an optional documentation string, the expected value, and the actual value, respectively. If the latter two are not equal, an exception is raised.

When JUnit is applied in a "test-first" or "test-driven" approach, the objective is to create test cases before the corresponding functionality is implemented (Beck, 2002). In practice, code and test case development are very much intertwined: the developer may start creating a small initial test case, followed by an initial version of the corresponding class which is just good enough to make the test pass. From then on, in a series of small iterations, the test is extended a little such that it fails, the production code is extended to make the tests pass, and the process is repeated.

JUnit test cases that have been developed before the code are clearly the result of a *black box* testing strategy. In addition to that, developers can make use of *white box* testing techniques: After having implemented their method, they can analyze the code and see if, for example, their test cases indeed exercise both branches of every if–then–else statement. With an explicit testing criterion requiring, say, 90% branch coverage, white box testing techniques are built into the process. Thus, we believe that JUnit test cases are typically derived using a *gray box* test generation strategy, consisting of a mixture of black and white box testing activities. (Note that it is *very* unlikely that a test suite achieving 90% branch coverage can be derived in a purely test-first manner: see our discussion in Section 6.3).

[2] One can even argue that agile development has become popular thanks to the success of JUnit.

Note that the typical use of JUnit requires that the full test suite is executed at every small modification. A consequence of this expected use is that JUnit test cases are required to execute quickly.

5. 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 (Section 5.1) identify possibly relevant metrics, and turn this relevance into a hypothesis to be analyzed (Section 5.2). Subsequently, we describe test suite metrics (Section 5.3), as well as the statistical methods that can be used to evaluate the hypothesis (Section 5.4).

5.1. Object-oriented metrics

Which metrics are suitable candidates for characterizing the testability factors discussed in the previous section? We will use the metrics suite proposed by Binder (1994) as a 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 (1994), 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 Section 2 we discuss various alternative metrics indicative of testability. The metrics discussed below and used in our experiment have the advantage that they are easier to implement and understand.

5.1.1. Notation

To give each metric a concise and unambiguous definition, we use the following notation, which is based on the notation proposed by Briand et al. (1999).

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

- 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:

- $M_{\mathrm{In}}(c)$, the set of methods that c inherits.
- $M_{\mathrm{D}}(c)$, the set of methods that c newly declares, i.e., either as abstract method or with a method body. Inherited methods that are not overridden are not part of this set.
- $M(c) = M_{\mathrm{D}}(c) \cup M_{\mathrm{In}}(c)$ the set of methods of c,
- $M_{\mathrm{Im}}(c)$, the set of methods that c implements. For each of these methods, c provides a method body. Note that $M_{\mathrm{Im}}(c) \subseteq M(c)$.
- $M(C) = \cup_{c \in C} M(c)$, the set of all methods.

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

- $\mathrm{MI}(m)$, the set of methods invoked by m. $m' \in \mathrm{MI}(m)$ iff the body of m contains an invocation of method m'. We will assume that the type of the object for which m' is invoked is determined statically (as done, for example, by a Java compiler).

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

- $F_{\mathrm{In}}(c)$, the set of fields that c inherits.
- $F_{\mathrm{D}}(c)$, the set of fields that c newly declares.
- $F(c) = F_{\mathrm{In}} \cup F_{\mathrm{D}}$, the set of fields of c.
- $F(C) = \cup_{c \in C} F(c)$, the set of all fields.

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

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

5.1.2. Metrics

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

Depth of inheritance tree

$$\mathrm{DIT}(c) = |\mathrm{Ancestors}(c)|$$

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 with this requirement, while C++ does not.

Fan out

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

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

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

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

The FOUT metric we use is an adaptation of Chidamber and Kemerer's CBO metric. In a sense, FOUT is a one-way version of CBO; it only counts the number of classes that c

uses, not the classes it is used by. The definition of CBO follows from the definition of FOUT if uses(c, d) is replaced by uses$(c, d) \vee$ uses(d, c).

Lack of cohesion of methods

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

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

This definition of LCOM is proposed by Henderson-Sellers (1996). 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

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

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

$$\mathrm{NOC}(c) = |\mathrm{Children}(c)|$$

Number of fields

$$\mathrm{NOF}(c) = |F_{\mathrm{D}}(c)|$$

Number of methods.

$$\mathrm{NOM}(c) = |M_{\mathrm{D}}(c)|$$

Response for class

$$\mathrm{RFC}(c) = |M(c)up_{m \in M_{\mathrm{Im}}(c)}\mathrm{MI}(m)|$$

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

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

where VG(m) is McCabe's cyclomatic complexity number (Watson and McCabe, 1996) for method m. VG(m) is given by the size of the basis set of paths through the control-flow graph of function m. For a single-entry, single-exit control-flow graph consisting of e edges and n nodes, $\mathrm{VG}(m) = e - n + 2$.

5.2. Goal and hypotheses

We set up our experiments for evaluating this set of metrics using the GQM/MEDEA[3] framework proposed by Briand et al. (2002). 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 6.

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 associated with 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 test suite. Well-known cost models such as Boehm's COCOMO (Boehm, 1981) and Putnam's SLIM model (Putnam, 1978) 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 5.3 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 associated with the size of the corresponding test suite?

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

$\mathrm{H}_0(m,n)$: There is *no* association between object-oriented metric m and test suite metric n;

$\mathrm{H}_1(m,n)$: There is a association 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.

[3] Goal Question Metric/MEtric DEfinition Approach.

5.3. 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.

5.4. 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 (Siegel, 1998). 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. The value of r_s can range from -1 (perfect *negative* correlation) to 1 (perfect *positive* correlation). A value of 0 indicates no correlation. 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 (Siegel, 1998). This statistic assigns a significance p to the correlation found by taking the number of observations into account. The significance p indicates the probability that the observed value is a chance event given the number of data pairs. It allows us to reject $H_0(m,n)$ (and accept $H_1(m,n)$) with a level of confidence equal to $1 - p$.

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 all 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 (Siegel, 1998). This process is repeated for all the object-oriented metrics in our set, and finally the results are presented in a table (see Section 6).

6. Case studies

We considered five software systems in our experiments. The first four, DocGen, Jackal, Monitor and ZPC, are systems developed at the Software Improvement Group (SIG). Additionally, we included Apache Ant, an open source automation tool for software development. All systems are written in Java, and are unit tested at the class level by means of JUnit. Basic size measurements of all systems are presented in Table 1. The LOC measure used here is defined as the newline count of all .java files.

6.1. Systems of the SIG

DocGen is a documentation generator; it processes source code of other programs, and generates technical documentation based on facts that are contained in the source code. DocGen is described in more detail in van Deursen and Kuipers (1999). **Jackal** is a tool capable of transforming Cobol code. Its tasks are pretty printing and code structure transformations. **Monitor** is used as part of the Software Portfolio Monitor service offered by the SIG. It collects metrics and performs other analysis in order to provide an overview of the quality of a software system. **ZPC** is a small tool that extracts strings suitable for localization from Cobol files.

The development process at the SIG is based on Kent Beck's eXtreme Programming (XP) methodology (Beck, 1999). The use of *coding standards* has a clear implication for the structure of the code; methods are not allowed to exceed 20 lines, and cannot have more than seven parameters. Also, unused fields, parameters or exceptions have to be removed. These conventions are enforced automatically through the (mandatory) use of the Eclipse Java IDE.

Consistent with XP methodology, development at the SIG is test-driven. If possible, test cases are written before

Table 1
Size figures for the studied systems

	LOC	Classes	Tested classes
DocGen	88,672	640	138
Jackal	14,355	134	59
Monitor	10,269	124	55
ZPC	9599	90	43
Ant	172,906	887	111

Table 2
Coverage levels obtained for all systems

	Statement (%)	Conditional (%)
DocGen	81	73
Jackal	98	93
Monitor	97	93
ZPC	99	96
Ant	68	60

actual development is started. Such test cases then serve as a specification of the required functionality. In other words, tests written in this fashion are black-box tests. The programmers extend the test suites by white-box tests after initial implementation is finished.

The SIG systems considered for study differ with respect to the testing criteria used. DocGen, which was studied prior to the other systems, has not been tested using an explicitly defined testing criterion (except that each method must have at least one test case). The study of this system therefore contains an uncontrolled variable, possibly making results harder to interpret. In contrast, the Jackal, Monitor and ZPC systems were tested with the following testing criterion in mind:

- Statement coverage must be at least 90%.
- Include a test case for each execution path.

The statement coverage level obtained by the test suites is controlled using an automatic coverage tool, while programmers manually check the existence of test cases for each execution path. Table 2 shows the coverage levels obtained for each system. Note that Jackal, Monitor and ZPC score significantly higher than DocGen and Ant. We discuss issues pertaining to test suite quality in more detail in Section 6.3.

6.2. Apache Ant

Ant[4] is a build tool, and is being developed as a subproject of the Apache web server. The Ant source code is kept in a public CVS repository, from which we obtained the 1.5.3 branch, dated 16 April 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.

However, there is one major difference between the testing of Ant and the testing of DocGen. The developers of Ant did not start using the current testing procedure until late in the development process.[5] The DocGen development team applied their testing approach from the start of the development of DocGen. Hence, the classes of DocGen have been subjected to a more homogeneous testing effort.

6.3. Test suite quality

A common way to measure the quality of (white-box) test suites is to determine the test coverage, i.e., the extent to which a test suite exercises relevant elements of the code under test. The relevant code elements are ultimately pointed out by the testing criterion. For example, McCabe's structured testing criterion requires that a "basis set of paths" is exercised for each module. A meaningful test coverage measurement would then determine to what extent the test suite exercises a basis set of paths through each module (Watson and McCabe, 1996).

As mentioned before, the DocGen and Ant systems have not been tested with an explicit testing criterion in mind. Measuring the quality of the respective test suites is impaired by this lack of testing criteria, since it is hard to identify the code elements to be covered. However, we performed two basic test coverage measurements as sanity checks of the subject systems.[6] First, we determined the percentage of statements of a class exercised (thus, covered) by the test suite, giving rise to each class' *statement coverage*. The quality of the test suites for Jackal, Monitor and ZPC is controlled using the same measure of statement coverage. Second, a class's *conditional coverage* is given by the percentage of conditional branches exercised.

The coverage measurements were restricted to the classes that have associated test classes. The average coverage levels obtained for all systems are presented in Table 2.

To put these numbers in perspective, recall that neither the Ant nor the DocGen developers adopted an explicit testing criterion. Test suites developed without such a criterion, even those that are considered "excruciatingly complete", often do not achieve more than 60% statement or branch coverage (Binder, 2000, p. 358). This was illustrated, for example, by Horgan and Mathur, who analyzed the coverage of the publicly available test suites for TEX and AWK (Horgan and Mathur, 1992; Knuth, 1984). Although the test suites for both programs reflected careful analysis of functionality and significant field debugging, the state-

[4] Web: http://ant.apache.org.

[5] Based on personal communication.

[6] The actual measurements were performed using the Clover coverage tool (Web: http://www.cenqua.com/clover.).

ment coverage figures for AWK and $\TeX$ were 70% and 80%, respectively.

Furthermore, achieving 100% branch or statement coverage is often unattainable due to infeasible paths, dead code, and exception handling. Ten percent to 15% is a typical allowance for such anomalies in complex and mature systems (Binder, 2000, p. 359).

From this we conclude that the 70–80% coverage obtained for Ant and DocGen is consistent with the assumption that Ant and DocGen are well-tested systems. The coverage is better than the 60% expected for a system tested without an explicit adequacy criterion, and for DocGen not even that far from the 90% that is feasible at most in practice. The three SIG systems that were tested with 90% statement coverage in mind, i.e., Jackal, Monitor and ZPC, more than accomplish their testing goal.

7. Case study results

7.1. Data collection

In order to collect data from subject systems, we have used the Eclipse platform[7] to calculate the metrics. An existing plug-in for Eclipse, the "Eclipse metrics plug-in",[8] has been extended to calculate our set of metrics. We are in the process of making the implementations of our metrics available to the general public.

7.2. Results

The Jackal, Monitor and ZPC systems were merged into a single case study, that will be referred to as Jackal–Monitor–ZPC. They were developed using the same methodology, using the same testing criterion.

Tables 3–5 hold the results of the experiments described in Section 5 for DocGen, Jackal–Monitor–ZPC, and Ant, respectively. Each table contains the values of Spearman's rank-order correlation coefficient (r_s) on the left hand side, and the significance (p) as obtained by the t-test on the right hand side. The number of data points corresponds to the number of tested classes shown in Table 1. DocGen thus has 138 data points, Jackal–Monitor–ZPC has 157, and Ant has 111 data points.

The source code metrics which are significantly correlated to the test suite metrics at the 99% confidence level, are set in boldface in Tables 3–5. For these, we can reject $H_0(m,n)$ and accept $H_1(m,n)$. Note that the accepted (and rejected) hypotheses are the same for all systems, except for the LCOM and NOF metrics. LCOM is not significantly correlated for DocGen, while it is for Jackal–Monitor–ZPC in case of dNOTC and both dLOCC and dNOTC for Ant.

Table 3
Measurement results for DocGen

r_s	dLOCC	dNOTC	p	dLOCC	dNOTC
DIT	−.0368	−.0590	DIT	.670	.492
FOUT	.555	.457	**FOUT**	<.01	<.01
LCOM	.166	.207	LCOM	.0517	.0148
LOCC	.513	.518	**LOCC**	<.01	<.01
NOC	−.0274	.00241	NOC	.750	.978
NOF	.248	.233	**NOF**	<.01	<.01
NOM	.355	.401	**NOM**	<.01	<.01
RFC	.537	.520	**RFC**	<.01	<.01
WMC	.422	.460	**WMC**	<.01	<.01

Table 4
Measurement results for Jackal, Monitor and ZPC

r_s	dLOCC	dNOTC	p	dLOCC	dNOTC
DIT	.109	−.0635	DIT	.177	.434
FOUT	.481	.240	**FOUT**	<.01	<.01
LCOM	.112	.215	LCOM	.167	<.01
LOCC	.643	.547	**LOCC**	<.01	<.01
NOC	−.0488	−.0218	NOC	.548	.788
NOF	.191	.177	NOF	.0171	.0280
NOM	.369	.485	**NOM**	<.01	<.01
RFC	.569	.455	**RFC**	<.01	<.01
WMC	.440	.511	**WMC**	<.01	<.01

Table 5
Measurement results for Ant

r_s	dLOCC	dNOTC	p	dLOCC	dNOTC
DIT	−.0456	−.201	DIT	.634	.0344
FOUT	.465	.307	**FOUT**	<.01	<.01
LCOM	.437	.382	**LCOM**	<.01	<.01
LOCC	.500	.325	**LOCC**	<.01	<.01
NOC	.0537	−.0262	NOC	.575	.785
NOF	.455	.294	**NOF**	<.01	<.01
NOM	.532	.369	**NOM**	<.01	<.01
RFC	.526	.341	**RFC**	<.01	<.01
WMC	.531	.348	**WMC**	<.01	<.01

7.3. Discussion

What can we learn from the data collected from Tables 3 and 5? 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 Tables A.1–A.3. 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).

[7] Web: http://www.eclipse.org.

[8] Web: http://sourceforge.net/projects/metrics.

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 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.

7.3.1. 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 all systems (except for NOF in Jackal–Monitor–ZPC). 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.

7.3.1.1. 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), for Jackal–Monitor–ZPC it is insignificant, 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. We can therefore not claim that NOF is acting as a test case construction factor in general.

A possible explanation is given by the definition of the NOF metric. In Section 5 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 remains the subject of further research.

7.3.1.2. Number of methods (NOM). One would expect that the number of methods primarily affects the number of test cases to be written (given a common testing criterion of having at least one test case for each 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 and Jackal–Monitor–ZPC indeed lives up to this 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 A.3), 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 correlations between the NOM and NOF metrics (see Tables A.1 and A.2) is only moderate for DocGen and Jackal–Monitor–ZPC confirms this explanation.

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

We observe that the WMC metric correlates strongly with the NOM metric for all systems (see Tables A.1–A.3). Also, the relationships to the other metrics are very similar for both WMC and NOM. An explanation is offered by the fact that for all systems, the VG value of each method tends to be low, and close to the average. The averages, maximums and standard deviations of the VG values are shown in Table 6. 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 the SIG systems are a result of a coding standard in use at the SIG. According to the coding standard, each method should not contain more than 20 lines of code.

7.3.2. Inheritance-related metrics

The second group of metrics we consider deals with inheritance: DIT measures the superclasses (the depth of

Table 6
Summary statistics on McCabe's cyclomatic complexity number (VG)

	Avg.	Max.	Std. dev.
DocGen	1.31	17	.874
Jackal–Monitor–ZPC	1.74	11	1.17
Ant	2.14	61	2.91

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.

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., Binder, 2000), 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., Binder, 2000).

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.

7.3.3. External dependencies

The fan out (FOUT) and response for class (RFC) metrics measure dependencies on external classes (FOUT) and methods (RFC). In all case studies these metrics are significantly correlated with both test suite metrics.

7.3.3.1. 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 Jackal–Monitor–ZPC and Ant). The fan out of a class measures the number of other classes 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.

7.3.3.2. Response for class (RFC). From the definition in Section 5, 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 all systems. Given the correlations between the RFC metric and both the NOM and FOUT metrics, the observed correlations between the RFC and dLOCC metrics are as expected.

7.3.4. 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 significantly correlated with both test suite metrics for Ant, but not for DocGen or Jackal–Monitor–ZPC.

To understand why this is the case, observe that Tables A.1–A.3 show that for all 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, in-cohesive 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.

We conjecture that, given the correlations between NOF and LCOM, the divergence between the case studies can be explained. In the case of Ant, there is a moderate correlation between NOF and dLOCC, which is absent in the other systems. For DocGen the same correlation is rather weak, while for Jackal–Monitor–ZPC it is even insignificant. The correlation between LCOM and the test suite metrics for Ant (and the lack therefore for DocGen and Jackal–Monitor–ZPC) can therefore be caused indirectly by NOF, which is a plausible component of the LCOM metric as we saw above.

8. Threats to validity

8.1. Confounders

8.1.1. Class size

In Emam et al. (2001) the authors investigate whether class size (LOCC) has a possible confounding effect on the validity of object-oriented metrics (the C&K metrics, among others). It turns out that associations between the investigated metrics and fault-proneness disappear after class size has been taken into account. The authors therefore conclude that for their cases class size has a strong confounding effect.

This conclusion is not shared by Evanco (2003), which argues that introducing class size as an additional independent variable can result in models that lack internal consistency. For example, it would make no sense to correct for confounding effects of LOCC on the WMC metric, since the two are highly correlated (our results confirm that claim, see Tables A.1 and A.2). Furthermore, confounding effects by class size are shown to be illogical by a temporal argument. Class size is fixed only after development of the class is complete, while measures such as coupling (FOUT, RFC) and attribute counts (NOM, NOF) can already be determined during early development or even during the design phase.

Class size has not yet been shown to have a confounding effect in the context of testability assessment, and since evidence of such effects in other contexts (fault-proneness) is dubious, we do not consider class size to be a confounder in our experiment.

8.1.2. Testing criteria

A possible confounder for the validity of testability metrics is given by the testing criterion in use at the project under consideration. In a white-box testing context, a testing criterion determines what structural aspects should be tested, and consequently, a testing criterion determines the number (and nature) of required test cases. Therefore, testing criteria can cause associations between metrics counting structural aspects and the number of test cases actually implemented. We discussed in Section 6 that two out of five systems (DocGen and Ant) studied in the experiment do not follow explicitly stated testing criteria.

The validity of our results is threatened by this fact in two ways. First, it is hard to judge applicability of our results to other systems, since the testing criteria may be incompatible. Second, a missing testing criterion hinders evaluation of test suite quality. Therefore, test suite quality is uncontrolled in cases of DocGen and Ant. We did perform coverage measurements in order to show test suite quality is up to general standards, but it is unknown whether those measurements are compatible with the actual (not specified) testing criterion. A concern with the Ant and DocGen cases is therefore that the results may be confounded by effects caused by uncontrolled test suite quality.

In contrast, the remaining systems, i.e., Jackal, Monitor, and ZPC, are tested using an explicit testing criterion. Test suite quality is therefore not uncontrolled in those cases. Furthermore, the results obtained for all systems are largely the same, justifying the conclusion that the studied metrics are not merely measuring test suite quality.

8.2. Selection bias

In both systems we considered those classes that have associated test classes. A possible selection bias is therefore present. Classes which were considered "too hard to test" or "too simple to warrant a testing effort" might not have associated test classes at all, and are therefore not included in the data. For example, classes implementing graphical user interfaces are typically not included in automated test suites. We found this to be largely true for the DocGen test suite. An explanation is that test frameworks like JUnit are not suitable for testing graphical user interfaces. As a result, our experiment is limited to assessing validity of testability metrics within the context of the JUnit (or similar) testing framework(s). Classes which are inherently not suitable to be tested using such frameworks can then safely be ignored.

9. 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 five Java systems totalling over 290 KLOC. 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 commercial Java systems as case studies.

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 a range of five Java systems has already been studied, it is desirable to extend our findings to a larger number of systems, developed in different languages and using all sorts of different development methodologies. In particular the experiments should be repeated for systems tested using different test criteria than statement/conditional coverage.

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. Furthermore, it would be interesting to study how refactorings (either at the code level Fowler, 1999 or in the test cases van Deursen et al., 2002) would affect testability metrics.

Third, more metrics are proposed to measure testability at various levels of abstraction. For instance, Wheeldon and Counsell (2003) and Baudry et al. (2003) propose a number of metrics defined at the design level. These metrics could also be investigated empirically to validate their use.

Another opportunity 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 Mouchawrab et al. (2005) 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 and Per John from the Software Improvement Group for their 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 also would like to thank the anonymous SCAM and JSS reviewers for their useful suggestions. We thank Tom Tourwé, Tijs van der Storm, Jurgen Vinju, Vania Marangozova (all CWI) and Gerd Gross (TU Delft) for commenting on drafts of this paper. Partial support was received from ITEA (Delft University of Technology, project MOOSE, ITEA 01002).

Appendix A. Correlations between the source metrics

Tables A.1–A.3 provide the correlations among the source code metrics.

Table A.1
r_s values between the object-oriented metrics for 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

Table A.2
r_s values between the object-oriented metrics for Jackal, Monitor and ZPC

	DIT	FOUT	LCOM	LOCC	NOC	NOF	NOM	RFC	WMC
DIT	1								
FOUT	.0598	1							
LCOM	−.243	.0174	1						
LOCC	−.0877	.678	.349	1					
NOC	−.0426	−.149	.0500	−.0511	1				
NOF	.0441	.300	.636	.441	−.0500	1			
NOM	−.236	.333	.653	.770	.0874	.516	1		
RFC	−.0543	.783	.374	.923	−.0741	.467	.770	1	
WMC	−.202	.482	.552	.864	.0622	.509	.926	.851	1

Table A.3
r_s values between the object-oriented metrics for 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

References

Baudry, B., Traon, Y.L., Sunyé, G., 2002. Testability analysis of a UML class diagram. In: Proceedings of the Ninth International Software Metrics Symposium (METRICS02). IEEE Computer Society, pp. 54–66.

Baudry, B., Traon, Y.L., Sunyé, G., Jézéquel, J.-M., 2003. Measuring and improving design patterns testability. In: 9th IEEE International Software Metrics Symposium (METRICS 2003). IEEE Computer Society, pp. 50–59.

Beck, K., 1994. Simple smalltalk testing: with patterns. Smalltalk Report 4 (2).

Beck, K., 1999. eXtreme Programming eXplained. Addison-Wesley, Reading, MA.

Beck, K., 2002. Test-Driven Development: By Example. Addison-Wesley.

Beck, K., Gamma, E., 1998. Test infected: programmers love writing tests. Java Report 3 (7), 51–56.

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

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

Boehm, B., 1981. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ.

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

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

Chidamber, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

Dagpinar, M., Jahnke, J., 2003. Predicting maintainability with object-oriented metrics – an empirical comparison. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, pp. 155–164.

du Bousquet, L., Delaunay, M. (Eds.), Proceedings ISSRE International Workshop on Testability Assessment (IWoTA), IEEE Explore, 2004.

Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N., 2001. The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering 27 (7), 630–650.

Evanco, W.M., 2003. Comments on "the confounding effect of class size on the validity of object-oriented metrics". IEEE Transactions on Software Engineering 29 (7), 670–672.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Freedman, R., 1991. Testability of software components. IEEE Transactions on Software Engineering 17 (6), 553–564.

Henderson-Sellers, B., 1996. Object-Oriented Metrics. Prentice Hall, New Jersey.

Horgan, J.R., Mathur, A.P., 1992. Assessing testing tools in research and education. IEEE Software 9 (3), 61–69.

ISO, International standard ISO/IEC 9126, 1991. Information technology: software product evaluation: quality characteristics and guidelines for their use.

Jungmayr, S., 2002. Identifying test-critical dependencies. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, pp. 404–413.

Khoshgoftaar, T., 2001. Modeling software quality with classification trees. In: Pham, H. (Ed.), Recent Advances in Reliability and Quality Engineering. World Scientific, pp. 247–270.

Knuth, D.E., 1984. A torture test for TEX, Tech. rep., Stanford University, Stanford, CA, USA.

McGregor, J., Srinivas, S., 1996. A measure of testing effort. In: Proceedings of the Conference on Object-Oriented Technologies. USENIX Association, pp. 129–142.

Menzies, T., Stefano, J.D., Ammar, K., McGill, K., Callis, P., Chapman, R., Davis, J., 2003. When can we test less? In: Proceedings of the Ninth International Software Metrics Symposium (METRICS03). IEEE Computer Society, pp. 98–111.

Mouchawrab, S., Briand, L.C., Labiche, Y., 2005. A measurement framework for object-oriented software testability. Tech. Rep. SCE-05-05, Carleton University.

Muthanna, S., Ponnambalam, K., Kontogiannis, K., Stacey, B., 2000. A maintainability model for industrial software systems using design level metrics. In: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, pp. 248–257.

Nguyen, T.B., Delaunay, M., Robach, C., 2002. Testability analysis for software components. In: 18th International Conference on Software Maintenance (ICSM 2002). IEEE Computer Society, pp. 422–429.

Nguyen, T.B., Delaunay, M., Robach, C., 2003. Testability analysis applied to embedded data-flow software. In: 3rd International Conference on Quality Software (QSIC 2003). IEEE Computer Society, pp. 351–358.

Nikora, A.P., Munson, J.C., 2003. Developing fault predictors for evolving software systems. In: Proceedings of the Ninth International Software Metrics Symposium (METRICS03). IEEE Computer Society, pp. 338–349.

Putnam, L., 1978. A general empirical solution to the macrosoftware sizing and estimating problem. IEEE Transactions on Software Engineering 4 (4), 345–361.

Rountev, A., Milanova, A., Ryder, B.G., 2004. Fragment class analysis for testing of polymorphism in java software. IEEE Transactions on Software Engineering 30 (6), 372–387.

Siegel, S., Castellan Jr., N.J.C., 1998. Nonparametric Statistics for the Behavioral Sciences. McGraw-Hill Book Company, New York.

Traon, Y.L., Ouabdesselam, F., Robach, C., 2000. Analyzing testability on data flow designs. In: International Symposium on Software Reliability Engineering (ISSRE 2000). IEEE Computer Society, pp. 162–173.

van Deursen, A., Kuipers, T., 1999. Building documentation generators. In: Proceedings of the International Conference on Software Maintenance (ICSM'99). IEEE Computer Society, pp. 40–49.

van Deursen, A., Moonen, L., van den Bergh, A., Kok, G., 2002. Refactoring test code. In: Succi, G., Marchesi, M., Wells, D., Williams, L. (Eds.), Extreme Programming Perspectives. Addison-Wesley, pp. 141–152.

Voas, J., 1992. PIE: a dynamic failure-based technique. IEEE Transactions on Software Engineering 18 (8), 717–727.

Voas, J., Miller, K., 1995. Software testability: the new verification. IEEE Software 12, 17–28.

Watson, A., McCabe, T., 1996. Structured testing: a software testing methodology using the cyclomatic complexity metric. T. NIST Special Publication 500–235, National Institute of Standards and Technology, Washington, DC.

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