
Proceedings Paper:
Simons, A.J.H. and Thomson, C.D. (2008) Benchmarking effectiveness for object-oriented unit testing. In: 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW'08. ICSTW '08, 09-11 Apr 2008, Lillehammer, Norway. IEEE, pp. 375-379. ISBN: 978-0-7695-3388-9.

https://doi.org/10.1109/ICSTW.2008.10

Benchmarking Effectiveness for Object-Oriented Unit Testing

Anthony J. H. Simons and Christopher D. Thomson
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello, Sheffield S1 4DP, United Kingdom
{A.Simons, C.Thomson}@dcs.shef.ac.uk

Abstract

We propose a benchmark for object-oriented unit testing, called the behavioural response*. This is a normative set of state- and equivalence partition-based test cases. Metrics are then defined to measure the* adequacy *and* effectiveness *of a test set (with respect to the benchmark) and the* efficiency *of the testing method (with respect to the time invested). The metrics are applied to expert manual testing using JUnit, and semi-automated testing using JWalk, testing a standard suite of classes that mimic component evolution.*

1. Introduction

The popularity of automated test execution tools, such as *JUnit* [1], has done much to raise awareness of the importance of frequent testing. However, there is little quantitative data about how effective testing with *JUnit* is in practice, since test-sets are constructed intuitively and are not evaluated for their quality. In other testing approaches, test coverage metrics are frequently cited, such as statement, branch or multiple-condition coverage [2]. However, it has been argued that these, and similar metrics, merely measure the amount of testing effort, rather than demonstrate test effectiveness [3]. For example, no guarantees can be made after testing about the quality of the tested software, where any remaining errors might be found, nor how serious they might be.

In the related discipline of software measurement, two schools of thought have emerged regarding how metrics should be derived. One school, exemplified by the Chidamber and Kemmerer metrics for object-oriented software [4], seeks to find easily measurable quantities, without making strong claims about what software qualities these indicate. The other school is exemplified by Basili et al.'s GQM approach, in which the conceptual goals are outlined, before determining operational questions and suitable quantitative metrics [5]. Though the latter approach is initially harder, the extra effort is worthwhile, since the resulting metrics more accurately measure the desired qualities.

We argue in this paper that many current testing metrics are like the former approach, in that they count easily measured quantities, without being able to make any strong claims about the quality of the test-set or the tested software. For example, path coverage metrics [6] can be fooled by unreachable dead code and, even if full path coverage is attained, the software has only been fully *exercised*, without verifying its *correctness*. Similarly, counting the number of exceptions raised [7] in unit testing may just indicate violated preconditions, rather than actual software faults. Test-set reduction approaches that seek to improve testing efficiency by filtering randomly generated test-sets [8, 9] divert attention away from measuring test *effectiveness* to measuring testing *effort*, a different quality.

As an example of something that is closer to the kind of metric we seek, mutation testing is able to characterize test-sets according to the number of faults that they detect in fault-seeded code [10]. To the extent that the seeded faults (code mutations) are representative of actual software faults likely to occur in the live system, the mutant-killing index is a true measure of test effectiveness. In the rest of this paper, we argue for testing metrics that measure test *effectiveness*, that is, which measure to what degree a given test-set can guarantee the correct behaviour of an implementation; and then which measure how *efficient* the testing method is in identifying effective tests.

2. Effective Testing

We believe that the basis for all *effective testing* metrics should be conformance testing, which measures how far the tested component behaves as intended, up to the testing assumptions. The notion of correctness is usually defined relative to a specification, whether this is expressed as an algebra [11], a finite-state automaton

[3] or a set of logical schemas [12]. The software is tested for conformance to the specification, which is assumed to be correct, predicting all significant paths and test outcomes. Since it is desirable to make *effective testing* accessible to the widest possible community, we suggest that the notion of a specification be broadened to include semi-formal maps from inputs to outputs, simple state machine sketches [13] and even the oracles devised by programmers in manual *JUnit* tests [1]. All of these ways of testing for conformance should be measurable by the proposed metrics.

2.1. Measuring degrees of correctness

The domain chosen for this paper is object-oriented unit testing. We assume here that the granularity of each unit is a single compiled class, whose constructors and methods may be invoked by the test harness and supplied with suitable test arguments. Test objects may be created, updated in various ways and inspected for conformance to a nominal specification. The reaction of an object may, at different times, be contingent upon its encapsulated state variables, or upon the input values supplied to its methods, or upon both.

Below, we seek to quantify the notion of a class's *behavioural response*, a normative test set. A measure of *test effectiveness* is defined on test sets in relation to their coverage of the behavioural response. A test set which exactly covers this is 100% effective, whereas a test set which fails to observe certain reactions, or which observes the same reaction many times, is less effective. The metric for effectiveness is calculated from collections of test cases:

BR, the behavioural response, a set of ideal tests;
T, a bag of tests to be measured for effectiveness;
$T_E = BR \cap T$, the set of effective tests in T;
$T_R = T - T_E$, the bag of redundant tests in T.

The collections T and T_R are bags, since they may contain duplicates, or equivalent tests that confirm the same property. Two metrics are now defined for test *effectiveness, Ef(T),* and test *adequacy, Ad(T)*:

$Ef(T) = (|T_E| - |T_R|) / |BR|$ test effectiveness
$Ad(T) = |T_E| / |BR|$ test adequacy

Effectiveness, the stronger metric, is the number of effective tests minus redundant tests as a proportion of the behavioural response. The weaker *adequacy* metric ignores redundant tests and is the number of effective tests as a proportion of the behavioural response.

2.2 Determining the behavioural response

The behavioural response of an object-oriented class is determined by considering its state-dependent and argument-dependent reactions separately. A *reaction* is any distinct behaviour, typically a unique path through a method (or constructor), which is intended by the designer, or specified in the requirements. We assume that the set of desired reactions is known.

Every method of the class has an *input response*, that is, a number of distinct reactions that depend on input values supplied to constructors and methods. We take the *maximum* possible number of reactions (over all states) as the input response for an operation, since certain states may not permit all reactions. Selecting inputs to trigger the input response is similar to finding exemplar values in equivalence partition testing [12].

The input response for the class is the sum of the input responses of its constructors and methods. The class also has a *state response*, a number of distinct reactions that depend on the state variables of the class. We take *any* distinct state-related reaction (over all operations and inputs) as indicating an abstract state. Selecting method sequences to reach all states is similar to computing the state cover in finite-state testing approaches [3].

We define the *behavioural response* of the class as the product of its state and input responses, that is, the reaction of the class to every input partition for every public operation in each of its abstract states. For example, if a class has 3 states and 5 public operations, of which 3 exhibit 2 input responses and the rest just 1, then the behavioural response has $3 \times (3 \times 2 + 2 \times 1) = 24$ test cases.

2.3 Relaxing testing assumptions

The *behavioural response* is a minimal test-set that can confirm, or rule out, every reaction in every state. It is still an idealized test set, in that it makes certain assumptions about the tested unit, in order to keep the size of the test set to a bare minimum. Note that this test set is already more efficient than tests intuitively chosen by programmers [13], which tend to over-confirm expected behaviour, but fail to rule out all unwanted behaviour [14, 15].

The two main assumptions are that the tested class has no redundant states; and that the boundaries are drawn correctly between the input partitions. These assumptions may be relaxed by extending the test set. We define $BR(S,P)$ as the *parametric behavioural response* in which S and P are integers denoting the level of testing needed to validate redundant states and

boundary values. For example, *BR(3,2)* tests method sequences up to length 3 in every state, to rule out faults in duplicated states hiding in shorter sequences [3]; and picks 2 exemplars from each partition, to better verify the partition boundaries [12]. The point of a benchmark parameterised in terms of transition path length and the number of partition exemplars is that it is possible to quantify *precisely* what any test set has tested, which makes comparisons possible between testing methods [14, 15].

3. Effectiveness of two testing methods

Two testing methods, using *JUnit* [1] and *JWalk* [16], were evaluated according to the *effectiveness* and *adequacy* metrics (see 2.1). In each case, the *time taken* to devise the test sets was also measured. A suite of 6 classes was tested, consisting of three related pairs, representing evolving software designs. The first pair was a linked and bounded implementation of a *Stack* (a change of implementation). The second pair was a loanable and reservable library *Book* (an extension by inheritance). The last pair was a basic and modified bank *Account* (with added preconditions on inputs). The *Stack* examples were biased towards state-related reactions, whereas the *Account* examples were biased towards input partition-related reactions; and the *Book* examples had both kinds of reaction.

Table 1. Size of the behavioural response

Test class	*API*	*input R*	*state R*	*BR(1,1)*
Stack1	6	6	2	12
Stack2	7	7	3	21
Book1	5	5	2	10
Book2	9	10	4	40
Account1	5	6	2	12
Account2	5	9	2	18

The behavioural response (see Table 1) of each test class was determined manually from the input response of each constructor and method and the state response of the class, by inspection of the source code.

3.1 Testing with JUnit

JUnit is the most widely used testing tool in the agile programming community [1]. Tests are manually constructed, according to the programmer's intuitions, and automatically executed. Tests are grouped into suites that exercise a particular method of the test class. Because of this localised focus, it is common to find redundant checks for the same property across different test suites. Another feature of *JUnit* is the encouragement to retest modified classes using the saved tests as regression tests. This promotes test reuse, where generating all-new tests from scratch is often needed to achieve the same coverage [14].

For this experiment, an expert software tester with 3 years' experience of using *JUnit* was asked to "test each behavioural response of each test class, similar to the transition cover and all equivalence partitions for method inputs". He was given full access to the source code, as well as *javadoc* specifications of each API. The time taken to create the test suites was logged.

Table 2. JUnit adequacy and effectiveness

Test class	*T*	T_E	T_R	*Ad*	*Ef*	*time*
Stack1	20	12	8	1.0	.33	11.31
Stack2	23	16	7	.76	.43	14.00
Book1	31	9	22	.90	-1.30	11.00
Book2	104	21	83	.53	-1.55	20.00
Account1	24	12	12	1.0	0.0	14.37
Account2	22	17	5	.94	.67	08.44

In Table 2, the size of the test-set *T* was determined by inspecting the expert's test-code and counting each assertion or forced exception as a distinct test case. The tester was able to create *adequate* tests, or nearly so, in most cases (but failed to find all states of *Book2*; and omitted 5 cases in *Stack2*). Most startling is the degree of redundant testing, giving low *effectiveness* scores. Considerable time was spent creating the tests; shading in Table 2 indicates cases that required extra time to debug faults in the test harness code.

3.2 Testing with JWalk

JWalk is a *lazy systematic unit-testing* tool [16]. The lazy systematic testing method is based on *lazy specification*, inferring a specification on-the-fly by semi-automated dynamic analysis of the evolving code, and *systematic testing*, generating complete test-sets that validate the state-space of a test class exhaustively to bounded depths. The *JWalk* tool allows the tester to compile a test oracle interactively, confirming key properties of the test class. Oracle values are re-used predictively during automated testing, which verifies the states and transitions of the test class exhaustively.

Although *JWalk* automates the exploration of states, it relies on programmer-adapted *generators* to supply specific exemplars from equivalence partitions, if these are desired. This might make it harder to test all input partitions. Elsewhere, *JWalk* removes the burden of having to think up unique test cases, by virtue of its

systematic exploration of states, so it was expected that the time taken to devise effective tests with *JWalk* might be much shorter than with *JUnit*. The human tester was given identical instructions to the above.

Table 3. JWalk adequacy and effectiveness

Test class	*T*	T_E	T_R	*Ad*	*Ef*	*time*
Stack1	12	12	0	1.0	1.0	0.42
Stack2	21	21	0	1.0	1.0	0.50
Book1	10	10	0	1.0	1.0	0.30
Book2	36	36	0	.90	.90	0.46
Account1	12	12	0	1.0	1.0	1.17
Account2	17	17	0	.94	.94	16.10

In Table 3, the test set *T* was determined by instructing *JWalk* to verify all states and transition paths of length 1. For *Account1*, the tests were run twice, using different generators (to vary the pattern of inputs); and for *Account2*, a new *BadIndexGenerator* had to be written (to deliberately raise the exceptions) and the tests were run three times. This explains the much longer time spent on the last example.

The state and transition coverage of *JWalk* was complete. The missed input partition cases were (in *Book2*) permutations in which a *Book* was issued and reserved by the same person; and (in *Account2*) where the *Account* was constructed with a negative balance. The outstanding result is the huge time-saving with *JWalk*: for most examples, it took seconds, rather than minutes, to confirm the unique test cases.

3.3 Conclusions on the Comparison

Semi-automated testing using *JWalk* is clearly more effective and massively more efficient than expert manual testing with *JUnit*. This is because the tool takes responsibility for exhaustive state and transition coverage, but makes best use of the human tester to confirm key oracle values interactively.

4. References

[1] Beck, K., *The JUnit Pocket Guide, 1st edn.,* O'Reilly, Beijing, 2004.

[2] BSI, *BS-7925-2 Software Component Testing, Draft 3.4,* http://www.testingstandards.co.uk/Component Testing.pdf, BSI/British Computer Society, London, 2001.

[3] F. Ipate, and W. M. L. Holcombe, "Specification and testing using generalised machines: a presentation and a case study", *Softw. Test., Verif. Reliab. 8(2),* John Wiley, Chichester, 1998, 61-81.

[4] S. Chidamber, and C. Kemmerer, "A metrics suite for object-oriented design", *IEEE Trans. Softw. Eng., 20(6),* IEEE Computer Society, Los Alamitos, 1994, 476-493.

[5] V. R. Basili, G. Caldiera, and D. Rombach, "The goal question metric approach", in: ed. J. Marciniak, *Encycl. Softw. Eng*., John Wiley, New York, 1994, 528-532.

[6] Information Processing Ltd., *Cantata++ for testing C, C++ and Java*. http://www.ipl.com/. IPL, Bath, 2008.

[7] C. Csallner, and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java", *Softw. Pract. Exp., 34 (11),* John Wiley, Chichester, 2004, 1025-1050.

[8] T. Xie, D. Marinov, and D. Notkin, "Rostra: a framework for detecting redundant object-oriented unit tests", *Proc. 19th IEEE Conf. Autom. Softw. Eng.,* IEEE Computer Society, Washington DC, 2004, 196-205.

[9] T. Xie, and D. Notkin, "Tool-assisted unit test selection based on operational violations", *Proc. 18th IEEE Int. Conf. Autom. Softw. Eng.* IEEE Computer Society, Montreal, 2003, 40-48.

[10] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system", *Softw. Test., Verif. Reliab. 15(2),* John Wiley, Chichester, 2005, 97-133.

[11] H. Y. Chen, T. H. Tse, and T. Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels", *ACM Trans. Softw. Eng. Meth., 10 (1),* ACM, New York, 2001, 56-109.

[12] P. A. Stocks, and D. A. Carrington, "A framework for specification-based testing", *IEEE Trans. Softw. Eng., 22(11),* IEEE Comp. Soc., Los Alamitos, 1996, 777-793.

[13] W. M. L. Holcombe, "Where do unit tests come from?", *Proc. 4th Int. Conf. on Extreme Progr. and Flexible Proc. in Softw. Eng., LNCS 2675*, Springer Verlag, Genova, 2003. 161-169.

[14] A. J. H. Simons, "A theory of regression testing for behaviourally compatible object types", *Softw. Test., Verif. Reliab., 16,* John Wiley, Chichester, 2006, 133-156.

[15] A. J. H. Simons, and C. D. Thomson, "Lazy systematic unit testing: JWalk versus JUnit", *Proc. 2nd Test. Acad. Ind. Conf. Pract. Research Tech.*, IEEE Computer Society, Windsor, 2007, 138.

[16] A. J. H. Simons, "JWalk: a tool for lazy systematic testing of Java classes by introspection and user interaction", *Autom. Softw. Eng., 14(4),* Springer, USA, 2007, 369-418.