# What Do The Asserts in a Unit Test Tell Us About Code Quality? A Study on Open Source and Industrial Projects

Mauricio Finavaro Aniche, Gustavo Ansaldi Oliva, Marco Aurélio Gerosa
*Department of Computer Science*
*Institute of Mathematics and Statistics*
*University of São Paulo (USP) - Brazil*
*E-mail: {aniche, goliva, gerosa}@ime.usp.br*

***Abstract*—Unit tests and production code are intrinsically connected. A class that is easy to test usually presents desirable characteristics, such as low coupling and high cohesion. Thus, finding hard-to-test classes may help developers identify problematic code. Many different test feedbacks that warn developers about problematic code were already catalogued. In this paper, we argue that analyzing assert instructions in unit tests also aid in identifying and reasoning about potentially problematic pieces of code. We report an analysis conducted with both open source and industry projects relating assert instructions in a unit test with quality measures of the code being tested. We observed that when a production method has a unit test that uses the "assert" instruction in more than one object instance, it often exhibits higher cyclomatic complexity, number of lines of code, or higher number of method invocations. It means that developers should monitor the number of asserts in a unit test as it may indicate problems in the production code.**

***Keywords*-unit testing; unit test feedback; code quality; code smells; mining software repositories.**

## I. Introduction

Designing good, flexible, and maintainable systems is not easy. It is common that, after some time, the system design loses quality and maintenance becomes hard and expensive. In order to avoid the rot of the code, the software development community has been investing in some set of practices, such as pair programming, code review, and so on.

Unit tests constitute a key element of modern software engineering. Besides aiding in the reduction of bugs, it can also support the identification of problematic pieces of code [14]. Agile methodologies practitioners, for example, state that unit tests are a way to validate and improve class design [1]. The main argument is that code with low testability is likely to contain design bad practices. That is why the practice of Test-Driven Development (TDD) [4] is becoming popular, as developers work on class design incrementally, taking into account the feedback that the test (that is written before the production code) can give.

We highlight the intrinsic connection that exists between unit tests and the associated production code. According to Feathers [12], there is a great synergy between a testable class and a well-designed class. In fact, a testable class presents a few desirable characteristics, such as the ease to invoke its behaviors, simplicity in methods' pre and post-conditions, and the explicit declaration of all its dependencies [14]. McGregor [26] states that the act of writing a unit test can become very complex when the interactions between software components grow out of control. Hence, we argue that analyzing units tests and the feedback they give can provide developers with useful information about the associated production code.

In fact, our group has been studying the relationship between unit tests and the quality of the code being tested. We found out that the test can actually warn developers about design problems related to coupling, cohesion, or complexity. Many tests for a single production method or class, huge effort to write the scenario for the test, and the excessive use of mock objects are examples of how a test warns developers about bad design decisions [2] [3].

In this paper, we aim to understand the relationship between a specific part of the unit test and the quality of the associated code being tested. We are focusing on a piece of code that a unit test usually has: the output validation. To achieve that, unit tests make use of *assert* instructions, which match the expected result with the calculated one.

A test method that contains many different *assert* instructions is usually hard to understand [32] [29]. Based on what was mentioned before, we can infer that complex unit tests tend to be an effect of a complex production class. Therefore, a test method with many different asserts can be a consequence of a complex production class. Confirming such hypothesis would give developers another cheap way to validate their class design and, thus, refactor the system and ultimately reduce maintenance cost.

To better understand the phenomenon, we present an empirical study that makes use of mining software repository techniques applied to 19 open-source projects from the Apache Software Foundation [28] and 3 industry projects. We found out that there is no relationship between the quantity of the asserts in a unit test and the production method being tested. However, when counting not the number of asserts, but instead the number of objects that are being

asserted in a single unit test, we noticed that production methods that are tested by unit tests with this characteristic present a higher cyclomatic complexity, more lines of code, or do more method invocations than methods in which their unit tests assert only a single instance of object.

## II. Unit Testing and Asserts

A unit test is a piece of code written by a developer to exercise a very small, specific area of functionality of the code being tested [18]. For Java, the most popular unit testing framework is JUnit[1].

The structure of a unit test is divided into 3 parts: first, the developer sets up the specific scenario s/he wants to test; then, the developer invokes the behavior under test; finally, s/he validates the expected output. In Listing 1, we show an example of a unit test that tests the behavior of the *calculateTaxes()* method from a fictitious *Invoice* class.

JUnit offers a set of instructions that can be used to validate the expected result. These instructions are called *asserts*. The provided assert instructions enable developers to make several kinds of verification. For example, *assertEquals()* checks whether the produced output is equal to the expected value; *assertFalse()* checks whether the produced output is false. Even exceptions throwed can be validated through an *"expected"* parameter.

There are no limits for the number of asserts per test. A test can have zero asserts (not common, but in this case, the test would fail only if the tested code throws an exception), one assert (checks only one produced output), or more than one assert (checks many different resulting outputs). This is exactly what we plan to evaluate: if the quantity of asserts in a unit test is somehow related to the quality of the production method that is being tested.

```
class InvoiceTest {
  @Test
  public void shouldCalculateTaxes() {
    // (i) setting up the scenario
    Invoice inv = new Invoice("Customer", 5000.0);
    // (ii) invoking the behavior under test
    double tax = inv.calculateTaxes();
    // (iii) validating the output
    assertEquals(5000 * 0.06, tax);
  }
}
```

Listing 1. An example of a unit test

## III. Research Design

In order to verify whether there is a relationship between the quantity of asserts and production code quality, we conducted a study on 22 projects, being 19 of them from the Apache Software Foundation[2] and 3 from a software development company located in São Paulo, Brazil.

For each project, we extracted information about the assert instructions in all unit tests, such as the quantity of assert instructions and object instances they validate. Furthermore, we calculated code metrics in all production code that is being tested by the test suite. Afterwards, we ran statistical tests to check the hypotheses. Additionally, we did a qualitative analysis to understand the reasons for writing more than one assert per test.

The sub-sections below explain all the research design decisions. All data, scripts, and code used during this study are publicly available[3].

### A. Hypotheses

Driven by the idea that a unit test with many different asserts can be the symptom of a complex production method, we elaborated a set of hypotheses that relate the quantity of asserts to code quality. To measure code quality, this work relies on well-known code metrics, such as McCabe's cyclomatic complexity [25], Lines of Code [10], and number of method invocations [31]. The specific null hypotheses we intended to test were the following:

- *H1: Code tested by only one assert does not present lower cyclomatic complexity than code tested by more than one assert.*
- *H2: Code tested by only one assert does not present fewer lines of code than code tested by more than one assert.*
- *H3: Code tested by only one assert does not present lower quantity of method invocations than code tested by more than one assert.*

### B. Data extraction

In order to calculate the differences between code that is tested by only one assert and code that is tested by more than one, we needed to extract some information from all unit tests.

Given a simple unit test, such as the example in Listing 1, which tests the method *calculateTaxes()* from a fictitious *Invoice* class, the following information is extracted:

1) **The number of asserts:** In the example, there is just one assert instruction. Although not present in Listing 1, we consider that the JUnit instruction *@Test(expected=...)* also counts as 1 assert. Such instruction is employed to when one wants to make sure an exception was thrown.
2) **The production method that is being tested:** It refers to the production method a unit test is validating. We look for all method invocations inside the test and capture their belonging class. If the name of such class matches the prefix of the test class name, then

[1]http://www.junit.org. Last access on January, 30th, 2012.

[2]http://www.apache.org. Last access on 13th, January 2012.

[3]http://www.ime.usp.br/~aniche/msr-asserts/. Last access on 30th, April 2012.

we consider that the method invocation represents a production method being tested. As an example, in Listing 1, the test invokes the *calculateTaxes()* method from an instance of the class *Invoice*. This class name matches the prefix of the test class name, i.e., *Invoice* matches the prefix of *InvoiceTest*. Therefore, we consider that the method *calculateTaxes()* is indeed being tested by this particular unit test. Otherwise, it is just a support method that is not the target of this test. More than one production method can be tested by a single unit test.

### C. Supporting Tool

Current tools that calculate code metrics require the input to be in *byte code* form. However, when trying to run different metrics over different repositories, byte code is not always available. Therefore, we developed a tool to parse Java source code, recognize unit tests, and extract the information needed. This tool currently supports tests written in JUnit 3 and 4. The tool was written in Java, has been unit tested, and it is publicly available[4].

The tool is able to process unit tests that contain the following common characteristics: (i) create their own scenarios by instantiating objects and setting their respective attributes with pre-defined values, (ii) execute an action by invoking the method being tested, and (iii) checks the output using assert instructions. The unit test in Listing 1 is an example of a test that is correctly parsed by our tool. There are a few variations that are not currently supported by our tool. They are described in threats to validity (Section VIII).

*1) Code Metrics:* Most of the implemented algorithms are well-known in software engineering community. All metrics were calculated in method-level.

- **Cyclomatic Complexity (CC)**: The tool calculates McCabe's cyclomatic complexity [25]. A simple explanation of the algorithm would be that, for each method, a counter gets increased every time an *if, for, while, case, catch, &&,* ||, or *?* appears. In addition, all methods have their counter initialized as 1.
- **Number of Method Invocations**: It counts the number of method invocations that occur inside a method. It recognizes method chaining and adds one for each invocation. For example, the piece of code *a().b().c()* counts as 3 method invocations. In particular, this metric is similar to the one proposed by Li and Henry [31], which counts the quantity of method invocations in a class.
- **Lines of Code (LoC)**: The tool counts the number of lines per method. It basically counts the number of line breaks found.

[4]http://www.github.com/mauricioaniche/msr-asserts. Last access on 15th, January, 2012

### D. Selected projects

We pre-selected all 213 open-source projects from the Apache Git repository. In addition, 3 projects from a Brazilian software development company were also pre-selected. The 3 industrial applications are web-based and follow the MVC pattern [9].

The reason we selected Apache projects is that they belong to different domain areas, and their perceived quality is well-known by the development community. The 3 industrial projects, besides the similarity with other industrial projects, were selected purely by availability.

In order to reduce the bias from the data, we decided to work only with software projects that fulfilled the following criteria:

- **Minimum test ratio**: The ratio of the *total number of unit tests* per *total number of production methods being tested* should be greater than or equal to 0.2.
- **Minimum of success in production method detection algorithm:** The algorithm should be able to identify the tested production method in more than 50% of all unit tests that contain at least one assert.

The constant 0.2 was chosen through observation of the *test ratio* distribution in all projects. It excludes around 80% of projects (only 58 projects presented a test ratio greater than 0.2). There was no specific reason to choose the threshold of 50% in the production method detection algorithm. We just preferred to eliminate projects that were not well recognized by the tool.

From all pre-selected projects, only 22 projects satisfied these criteria: *commons-codec, commons-compress, commons-lang, commons-math, commons-validator, cxf-dosgi, directory-shared, harmony, log4j, log4j-extras, maven-2, maven-doxia-sitetools, maven-enforcer, maven-plugins, maven-sandbox, maven-scm, rat, shindig, struts-sandbox*, and the three industrial projects, which we called *CW, CL,* and *WC*.

With the exception of Harmony, which contains almost a million lines of code, 7k classes, 60k methods, and 26k unit tests, the other projects contain from 4k up to 145k lines of code, 41 to 1574 classes, 250 to 7500 methods, and 65 to 1600 unit tests. Based on these numbers, we can state that different projects from different sizes were included in this study. In Table I, we show the precise numbers for each project.

### E. Data Analysis

After analyzing all unit tests, we obtained their number of asserts and the respective production method they were testing. For each project, we then separated production methods into two different groups: the first group (group A) contained all production methods that were invoked by at least one unit test that contained a single assert instruction, while the other group (group B) contained all production

Table I
DESCRIPTIVE ANALYSIS OF SELECTED PROJECTS

| Project | Lines of Code | Classes | Methods | Tests |
|---|---|---|---|---|
| rat | 4,582 | 107 | 459 | 117 |
| maven-enforcer | 5,273 | 62 | 432 | 92 |
| maven-doxia-sitetools | 5,797 | 41 | 254 | 65 |
| Industry CP | 5,820 | 135 | 763 | 208 |
| commons-codec | 9,885 | 77 | 427 | 383 |
| commons-validator | 10,512 | 131 | 744 | 270 |
| Industry WC | 10,745 | 951 | 1,211 | 552 |
| log4j-extras | 11,503 | 147 | 745 | 498 |
| cxf-dosgi | 11,920 | 185 | 708 | 168 |
| commons-compress | 16,425 | 127 | 937 | 299 |
| log4j | 27,499 | 354 | 2,387 | 621 |
| maven-2 | 35,676 | 346 | 2,182 | 498 |
| maven-scm | 46,804 | 820 | 3,048 | 698 |
| commons-lang | 48,096 | 211 | 2,222 | 2,046 |
| maven-sandbox | 53,763 | 646 | 4,062 | 1,067 |
| struts-sandbox | 54,774 | 1,001 | 6,220 | 2,291 |
| Industry CW | 68,690 | 1,043 | 6,897 | 2,354 |
| shindig | 76,821 | 999 | 6,513 | 2,623 |
| commons-math | 96,793 | 877 | 5,478 | 2,281 |
| maven-plugins | 120,801 | 1,574 | 7,452 | 1,605 |
| directory-shared | 144,264 | 1,147 | 5,866 | 3,245 |
| harmony | 915,930 | 7,027 | 59,665 | 26,735 |
| Minimum | 4582.0 | 41.0 | 254.0 | 65.0 |
| 1st Quartile | 10570.0 | 132.0 | 744.2 | 277.2 |
| Median | 31590.0 | 350.0 | 2202.0 | 586.5 |
| Mean | 81020.0 | 818.5 | 5440.0 | 2214.0 |
| 3rd Quartile | 65210.0 | 987.0 | 6132.0 | 2222.0 |
| Maximum | 915900.0 | 7027.0 | 59660.0 | 26740.0 |

Table II
DISTRIBUTION OF ASSERTS IN PROJECTS

| Project | Zero Assert | One Assert | More Than One Assert |
|---|---|---|---|
| commons-codec | 112 | 125 | 146 |
| commons-compress | 172 | 26 | 101 |
| commons-lang | 156 | 397 | 1,488 |
| commons-math | 416 | 719 | 1,132 |
| commons-validator | 69 | 29 | 170 |
| cxf-dosgi | 22 | 60 | 80 |
| directory-shared | 284 | 1,250 | 1,704 |
| harmony | 5,318 | 7,063 | 14,263 |
| log4j | 39 | 464 | 118 |
| log4j-extras | 33 | 398 | 67 |
| maven-2 | 67 | 249 | 175 |
| maven-doxia-sitetools | 5 | 9 | 51 |
| maven-enforcer | 39 | 24 | 29 |
| maven-plugins | 470 | 377 | 583 |
| maven-sandbox | 96 | 645 | 311 |
| maven-scm | 242 | 180 | 276 |
| Industry CW | 327 | 1269 | 758 |
| Industry CP | 74 | 66 | 68 |
| Industry WC | 220 | 242 | 90 |
| rat | 29 | 27 | 61 |
| shindig | 362 | 1,098 | 1,163 |
| struts-sandbox | 2,049 | 157 | 81 |

methods that were invoked by at least one unit test that contained two or more asserts.

The same production method could appear in both groups, since it can be tested by many different unit tests with different number of asserts. Unit tests with zero asserts were ignored. Also, we did not allow duplicates in each group. For instance, if a production method *foo* is invoked by two different unit tests that have only one assert each, then this method will appear only once in group A. In Figure 1, we present a diagram that depicts the production methods separation into groups.

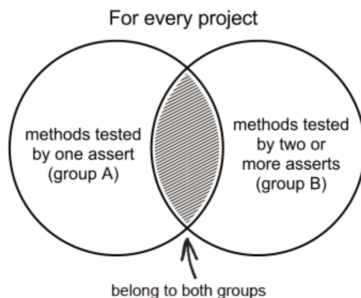


Figure 1. How we separated methods and their metrics in order to analyze the data

A significant difference in the values of the code metrics calculated for the two groups would indicate that the number of asserts may influence code quality. We thus executed the *Shapiro-Wilk test* to first check whether the data followed a normal distribution. As the data did not follow a normal distribution, the *Wilcoxon signed-rank test* was chosen. It is a non-parametric statistical hypothesis test used when comparing two samples to assess whether their population mean ranks differ. We then ran such test to compare both groups.

## IV. ASSERTS AND CODE QUALITY: RESULTS

In this section, we provide the results we obtained by investigating the relationship between the number of asserts and code quality.

### A. Descriptive Analysis

In Table II, we show the quantity of tests separated into three groups: zero asserts, one assert, and more than one assert. For each project, the group with the highest value is gray-colored. Observing the distribution of asserts per project, it is possible to see that projects contain a greater number of tests that make use of more than one assert: 15 projects presented this characteristic. The number of tests with one assert scored the highest value in only 7 projects. The struts-sandbox project presented a distinguishing high number of tests with zero asserts.

### B. Statistical Test

In Table III, we present the results of the Wilcoxon test (p-values) [5] we obtained for the difference in code quality

[5]The *commons-validator* project does not have enough data to run the statistical test.

Table III
RESULTING P-VALUES FROM APPLYING WILCOXON TEST TO THE GROUP OF ONE ASSERT AND THE GROUP OF MORE THAN ONE ASSERT

| Project | Cyclomatic Complexity | Method Invocations | Lines of Code |
|---|---|---|---|
| commons-codec | 0.1457 | 0.0257* | 0.2991 |
| commons-compress | 0.7899 | 0.5374 | 0.9122 |
| commons-lang | 0.6766 | 0.3470 | 0.8875 |
| commons-math | 0.9230 | 0.9386 | 0.9369 |
| commons-validator | 0.9477 | - | 0.9635 |
| cxf-dosgi | 0.8445 | 0.9567 | 0.9463 |
| directory-shared | 0.9518 | 0.1298 | 0.9972 |
| harmony | 0.0174* | 0.2822 | 0.5676 |
| log4j | 0.9489 | 0.6789 | 0.9885 |
| log4j-extras | 0.4811 | 0.6339 | 0.5703 |
| maven-2 | 0.2532 | 0.9490 | 0.4427 |
| maven-doxia-sitetools | 0.9561 | 0.9213 | 0.9595 |
| maven-enforcer | 0.9371 | 0.4064 | 0.9727 |
| maven-plugins | 0.9607 | 0.6946 | 0.9847 |
| maven-sandbox | 0.4277 | 0.6499 | 0.9133 |
| maven-scm | 0.9324 | 0.9846 | 0.9948 |
| Industry CW | 0.9999 | 0.2567 | 0.9999 |
| Industry CP | 0.2850 | 0.0003* | 0.4425 |
| Industry WC | 0.5380 | 0.5381 | 0.0561 |
| rat | 0.3162 | 0.4153 | 0.3263 |
| shindig | 0.9968 | 0.0252* | 0.9998 |
| struts-sandbox | 0.9758 | 0.2942 | 0.9649 |

when considering the quantity of asserts [6]. All p-values lesser than 0.05 indicate that there is a statistically significant difference between the observed groups, and the cells were gray-colored.

The numbers reveal that only one project in 22 had a statistical significant difference in cyclomatic complexity (H1). Three projects (13%) differ in the number of method invocations executed by the production method (H2). None differ in terms of lines of code (H3).

Therefore, **the quantity of asserts in a unit test is not related to the production code quality**. It means that developers that are trying to obtain feedback from their unit tests about the production code quality should not rely on the quantity of assert as an indicator.

We also tested some other metrics related to class design, such as LCOM [17] and Fan-Out [24]. However, no statistical significance was found. Furthermore, we checked for a correlation between number of asserts and code quality using the Spearman correlation formula, and no correlation was found.

## V. WHY MORE THAN ONE ASSERT?

As no relationship between the quantity of asserts and production code quality was found, we then decided to understand why a developer writes more than one assert per test. Thus, a qualitative study was conducted. The goal was to read the tests and categorize them according to the reason it needed more than one assert.

In this study, 130 tests were randomly selected from a population of around 128k tests. This sample size enables us to generalize the results with a confidence level of 95% and a confidence interval of 8%. After analyzing each selected unit test, we were able to separate them into the following categories:

- *More than one assert for the same object* (40.4%): When a method changes more than one internal attribute in the object or return an entirely new one, developers assert all attributes that should have changed by that method, using as many assert instructions to the same object (but to a different attribute) as needed.
- *Different inputs to the same method (38.9%)*: Some unit tests give different inputs to the same method and check all generated outputs. In these cases, developers usually instantiate the same class many times, and invoke the same method with a different input. In addition, sometimes the method being tested returns an instance of a new object, which must be validated. In both cases, developers write asserts to many different instance of objects.
- *List/Array (9.9%)*: When the method deals with a list or array, the test usually checks not only the size of the list, but also its internal content, validating the state of element by element.
- *Others (6.8%)*: Mock interactions, problems when trying to understand the unit test, and any other different reason found.
- *One extra assert to check if object is not null (3.8%)*: Some tests make an extra assert to check whether the returned object is not null.

As we somehow expected, the most common behavior in tests is to write more than one assert for the same object. The second most popular is to have different inputs to the same method. These two options combined represent almost all the individual unit tests investigated in this study: 40.4% and 38.9%, respectively. In Figure 2, we show the categories and their distribution over the selected sample.

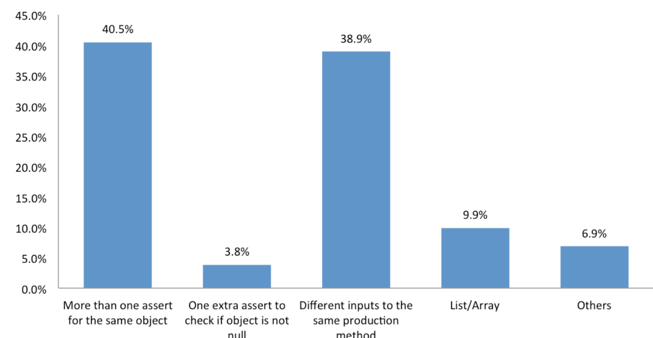


Figure 2. Why developers write more than one assert per test

[6]We decided not to use Bonferroni correction. It just does not make sense to try to combine all code metrics in order to affirm that a code is not well-written. Whether a piece of code presents many lines, or a high cyclomatic complexity, or a high number of method invocations, it can be considered as a piece of code that can be improved. Therefore, we analyzed each variable independently.

Inductively, one can notice that writing more than one assert for the same object can be a common behavior. Unit tests should assert the state of the entire object and, in order to achieve that, more than one attribute should be checked. Analogously, when testing a list, developers usually check the size of the list and each element that is contained inside of it.

However, trying different inputs to the same method in a single test does not make sense. We hypothesize developers thought it would just be too much work to test all those inputs in different tests. Hence, they preferred to instantiate the same object two, three, N times, and make asserts on all of them.

Consequently, we turned our attention to production methods whose unit tests contain assert instructions for more than one object instance. We decided to investigate whether there is a relationship between the number of object instances that are evaluated in assert instructions and code quality metrics. This study is discussed in the next section.

## VI. Asserted Objects and Code Quality

As mentioned before, some unit tests make asserts to more than one object instance. To better represent this case, we crafted the term *asserted object*, which refers to objects that are evaluated by assert instructions. In Listing 2, for example, we present a unit test from the Ant project that contains two asserted objects (*reportFile* and *reportStream*). In this specific case, the developer needed to make use of more than one object to completely test the behavior.

We suspect that they are related to the production code quality: a unit test that asserts more than one object instance may be a consequence of a bad production code.

```
public void testWithStyleFromDir() throws
    Exception {
    executeTarget("testWithStyleFromDir");
    File reportFile = new File(System.getProperty(
        "root"), "(path omitted)");
    assertTrue(reportFile.exists());
    assertTrue(reportFile.canRead());
    assertTrue(reportFile.length() > 0 );
    URL reportUrl = new URL(FileUtils.getFileUtils
        ().toURI(reportFile.getAbsolutePath()));
    InputStream reportStream = reportUrl.
        openStream();
    assertTrue(reportStream.available() > 0);
}
```

Listing 2. An example of a unit test from Ant project with two asserted objects

### A. Study Design

We improved our tool in order to also capture the number of asserted objects per unit test. The algorithm counts the number of object instances or primitive types asserted by the test. We do not distinguish between instances from different classes and instances from the same class (they are all subject to the counting).

Table IV
Distribution of Asserted Objects in Projects

| Project | Zero Assert | One Assert | More Than One Assert |
|---|---|---|---|
| commons-codec | 117 | 212 | 54 |
| commons-compress | 172 | 91 | 36 |
| commons-lang | 284 | 1,308 | 449 |
| commons-math | 568 | 1,247 | 453 |
| commons-validator | 69 | 131 | 68 |
| cxf-dosgi | 22 | 109 | 31 |
| directory-shared | 337 | 1,875 | 1,027 |
| harmony | 5,337 | 14,699 | 6,606 |
| log4j | 39 | 538 | 43 |
| log4j-extras | 33 | 419 | 46 |
| maven-2 | 67 | 348 | 77 |
| maven-doxia-sitetools | 5 | 46 | 14 |
| maven-enforcer | 39 | 46 | 7 |
| maven-plugins | 471 | 628 | 331 |
| maven-sandbox | 450 | 548 | 54 |
| maven-scm | 244 | 327 | 127 |
| Industry CW | 449 | 1,586 | 319 |
| Industry CP | 116 | 67 | 25 |
| Industry WC | 263 | 249 | 40 |
| rat | 29 | 57 | 31 |
| shindig | 506 | 1,625 | 492 |
| struts-sandbox | 2,049 | 202 | 36 |

We then collected data over the same projects, did the same separation in groups A and B, and ran the same statistical test presented before. The reformulated set of hypotheses is presented below:

- *H1: Code tested by only one asserted object does not present lower cyclomatic complexity than code tested by more than one assert.*
- *H2: Code tested by only one asserted object does not present fewer lines of code than code tested by more than one assert.*
- *H3: Code tested by only one asserted object does not present lower quantity of method invocations than code tested by more than one assert.*

### B. Descriptive Analysis

The numbers observed in Table IV are quite different from the ones regarding the quantity of assert instructions: all projects contain more tests with only one asserted object than tests with more than one asserted object.

### C. Statistical Test

In Table V, we can observe that six projects (27%) differ in cyclomatic complexity, four projects (18%) in the number of methods invocations, and five projects (22%) in terms of lines of code.

The results from the statistical test when looking to asserted objects were different too. In Table V, we observe that six projects (27%) presented a difference in cyclomatic complexity (as opposed to the single case in the previous test). The same happens with both the methods invocation (four projects, 18%) and the lines of code metrics (five

Table V
RESULTING P-VALUES FROM APPLYING WILCOXON TEST TO THE GROUP OF ONE ASSERTED OBJECT AND THE GROUP OF MORE THAN ONE

| Project | Cyclomatic Complexity | Method Invocations | Lines of Code |
|---|---|---|---|
| commons-codec | 0.0376* | 0.1376 | 0.0429* |
| commons-compress | 0.0385* | 0.0445* | 0.4429 |
| commons-lang | 0.9918 | 0.7207 | 0.9611 |
| commons-math | 0.0262* | 0.1873 | 0.0325* |
| commons-validator | 0.0329* | 0.6256 | 0.0223* |
| cxf-dosgi | 0.5351 | 0.9986 | 0.3919 |
| directory-shared | 0.9998 | 0.1408 | 1.0000 |
| harmony | 3.5300E-05* | 0.0001* | 0.0043* |
| log4j | 0.2749 | 0.6789 | 0.3574 |
| log4j-extras | 0.0658 | 0.6339 | 0.0594 |
| maven-2 | 0.2912 | 0.6228 | 0.1611 |
| maven-doxia-sitetools | 0.8012 | 1.0000 | 0.8986 |
| maven-enforcer | 0.2670 | 0.8219 | 0.1348 |
| maven-plugins | 0.1461 | 0.1972 | 0.0953 |
| maven-sandbox | 0.9821 | 0.1400 | 0.9815 |
| maven-scm | 0.9726 | 0.9193 | 0.9999 |
| Industry CW | 0.9999 | 0.0089* | 0.9986 |
| Industry CP | 0.9103 | 0.7909 | 0.3055 |
| Industry WC | 0.1274 | 0.4718 | 0.0606 |
| rat | 0.2213 | 0.1533 | 0.1373 |
| shindig | 0.0006* | 0.0238* | 0.0002* |
| struts-sandbox | 0.9994 | 0.6910 | 0.9995 |

projects, 22%). In summary, the number of significant results increased significantly.

We also tested some other metrics related to class design, such as LCOM [17] and Fan-Out [24]. However, no statistical significance was found for them. Furthermore, we checked for a correlation between the number of asserted objects and code quality using the Spearman correlation formula. Similarly, no correlation was found.

Based on this numbers, we can state that **production methods whose unit tests contain more than one asserted object may present worse code metrics than production methods that do not have this characteristic**.

### D. Measuring the difference

Besides analyzing the p-values, we also observed how big the difference is between these two groups. The Table VI depicts a descriptive analysis of the values for each metric calculated. The entries are given in the form *A/B*, representing the values for group A and group B respectively. We see that the difference in the average of both groups is not that big. In cyclomatic complexity, for example, the difference on average between both groups is almost one. In terms of lines of code and number of method invocations, only the *shindig* project presented a difference larger than 1. In a nutshell, although we found out a significant difference between both groups, such difference is not actually high.

## VII. DISCUSSION

In Table VII, we present a summary of our findings. As one can notice, when investigating the quantity of asserts, only a few projects presented a significant difference in production code quality when considering cyclomatic complexity, lines of code, and number of method invocations.

Table VII
SUMMARY OF THE RESULTS OF THE STATISTICAL TESTS: QUANTITY OF PROJECTS THAT REFUTED EACH HYPOTHESES

| Hypotheses | Qty of Projects (Asserts) | Qty of Projects (Asserted Objects) |
|---|---|---|
| H1 (Cyclom. Complexity) | 01 (05%) | 06 (27%) |
| H2 (Lines of Code) | 00 (00%) | 05 (22%) |
| H3 (Method Invocations) | 03 (13%) | 04 (18%) |

Based on what we observed in both studies, we can state that purely counting the number of asserts will not provide feedback about the production code quality. Instead, if developers count the number of objects that are being asserted in a unit test, that may provide useful information about the production method the unit test is testing. This feedback can help developers identify pieces of bad code. Even though the observed difference is not big, it is still important to keep managing complexity as it appears on software code [23].

A possible explanation for this phenomenon would be that a method that presents a higher cyclomatic complexity, many lines of code, or make many method invocations, contains many different possible paths of execution. Each path usually makes a different change into one or even many different objects. If a developer wants to test all the possible paths, s/he would spend too much time writing different unit tests. Because of that, s/he choose to try different inputs inside the same unit test.

What we learn from here is that if developers do not want to write many different unit tests to test different execution paths, it may be because of the high complexity of that method. Analogously, in Aniche's catalogue of test feedbacks [2], he showed that when developers write many unit tests for the same production method, it may be because of the complexity of such method.

Therefore, this pattern can be included in the test feedback catalogue. We suggest the **"More than One Asserted Object per Test"** pattern of feedback. When observing a unit test, developers should pay attention to the number of objects they are asserting.

## VIII. THREATS TO VALIDITY

### A. Construct Validity

The tool we developed has been unit tested, which gives some confidence about its external quality. However, as the number of different ways a programmer can write a unit test is almost infinite, there are some variations that are currently not supported by the tool. In particular, unit tests that contain asserts in private methods, in helper classes, or even in inherited methods, are currently not detected by the tool. In those cases, we discarded the tests from the analysis.

Table VI
DESCRIPTIVE ANALYSIS FROM THE NUMBERS USED IN THE STATISTICAL TESTS (THE ENTRIES ARE GIVEN A/B REPRESENTING THE VALUES FOR GROUP A AND GROUP B)

| Project | Median | Average | 1st Qu. | 2nd Qu. | 3rd Qu. | Min | Max |
|---|---|---|---|---|---|---|---|
| Cyclomatic Complexity | | | | | | | |
| commons-codec | 2.0 / 3.0 | 3.5735 / 3.9000 | 1.0 / 2.0 | 2.0 / 3.0 | 4.0 / 4.0 | 1.0 / 1.0 | 41.0 / 16.0 |
| commons-compress | 1.0 / 2.5 | 2.8548 / 2.7222 | 1.0 / 1.0 | 1.0 / 2.5 | 1.0 / 3.0 | 1.0 / 1.0 | 20.0 / 10.0 |
| commons-math | 1.0 / 2.0 | 2.6081 / 2.9939 | 1.0 / 1.0 | 1.0 / 2.0 | 1.0 / 3.0 | 1.0 / 1.0 | 33.0 / 38.0 |
| commons-validator | 1.0 / 1.0 | 1.9300 / 2.6562 | 1.0 / 1.0 | 1.0 / 1.0 | 1.0 / 1.0 | 1.0 / 1.0 | 11.0 / 11.0 |
| harmony | 1.0 / 1.0 | 2.3182 / 2.4903 | 1.0 / 1.0 | 1.0 / 1.0 | 1.0 / 1.0 | 1.0 / 1.0 | 77.0 / 86.0 |
| shindig | 1.0 / 2.0 | 2.5286 / 3.4265 | 1.0 / 1.0 | 1.0 / 2.0 | 1.0 / 4.0 | 1.0 / 1.0 | 37.0 / 37.0 |
| Lines of Code | | | | | | | |
| commons-compress | 6.0 / 6.5 | 10.6290 / 9.4444 | 2.0 / 2.0 | 6.0 / 6.5 | 11.0 / 11.0 | 2.0 / 2.0 | 60.0 / 60.0 |
| harmony | 3.0 / 3.0 | 7.34478 / 7.8988 | 2.0 / 2.0 | 3.0 / 3.0 | 7.0 / 7.0 | 0.0 / 0.0 | 231.0 / 223.0 |
| Industry CW | 4.0 / 2.0 | 6.9958 / 6.7671 | 2.0 / 2.0 | 4.0 / 2.0 | 8.0 / 2.0 | 1.0 / 2.0 | 60.0 / 60.0 |
| shindig | 3.0 / 6.0 | 9.6236 / 14.3080 | 2.0 / 2.0 | 3.0 / 6.0 | 9.0 / 15.0 | 0.0 / 2.0 | 184.0 / 184.0 |
| Number of Method Invocations | | | | | | | |
| commons-codec | 1.0 / 1.5 | 1.5609 / 2.1000 | 1.0 / 1.0 | 1.0 / 1.5 | 1.0 / 3.0 | 1.0 / 1.0 | 4.0 / 4.0 |
| commons-math | 2.0 / 2.0 | 2.5284 / 3.0779 | 1.0 / 1.0 | 2.0 / 2.0 | 3.0 / 3.5 | 1.0 / 1.0 | 12.0 / 21.0 |
| commons-validator | 2.0 / 1.0 | 1.8214 / 2.3571 | 2.0 / 1.0 | 2.0 / 1.0 | 2.0 / 1.0 | 1.0 / 1.0 | 6.0 / 6.0 |
| harmony | 2.0 / 2.0 | 2.7309 / 3.1127 | 1.0 / 1.0 | 2.0 / 2.0 | 3.0 / 3.0 | 1.0 / 1.0 | 39.0 / 38.0 |
| shindig | 4.0 / 7.5 | 6.9325 / 10.5789 | 2.0 / 3.0 | 4.0 / 7.5 | 7.0 / 18.0 | 1.0 / 1.0 | 36.0 / 36.0 |

The production method detection algorithm does not detect a few variations as well. The algorithm expects that the test class name prefix is always the same as the production class name. If that does not happen, then the algorithm is not able to make the link between the unit test and the production method. Furthermore, if the production method is invoked on a private or inherited method in the test class, inside the test setup method (a method that is executed before every unit test in that class), or even on a helper class, then the algorithm also does not detect it. On the other hand, a very common case is to refer to the tested type as an interface. This variation is detected by the algorithm: it looks to the concrete type and, consequently, finds the class that is actually being tested.

The mentioned limitations may exclude interesting projects from the analysis. The project filtering process may have also biased the sample, since the threshold for *minimum test ratio* may have leaded to the exclusion of interesting projects unnecessarily. A future work could be to calculate code coverage of tests and use it as a filter.

As suggested by the qualitative analysis, developers sometimes use the same unit test to check different inputs to the same production method. If a developer writes two variables that point to the same instance, our algorithm counts it as a test with two asserted objects. They may have polluted the data, and a future step would be to try to automatically remove these tests from the sample.

### B. Internal Validity

To reduce noise in the collected data, this study tried to filter all projects that could contain non-useful data. In this case, we removed projects in which the algorithm could not interpret more than 50% of its unit tests and did not contain an acceptable number of tests per method. That may be explained by the fact that Apache projects are usually complicated and, in order to test it, an integration test is more suitable than unit test. And that is why our algorithm did not recognize them.

The most effective way to relate a test to its production code would be to run the test case. However, when analyzing a huge quantity of projects at once, it becomes complicated to run all the tests automatically. However, the heuristic was based on a famous convention among developers. The number of tests that were recognized shows it. Therefore, we do not think that the heuristic would be a problem in practice.

The qualitative study involved only 130 tests, which is a small sample compared to the entire population. A more extensive qualitative study should be conducted in order to increase the confidence measure, further refine the conceived taxonomy, and more deeply understand the reasons a developer writes more than one assert.

The chosen metrics may have also influenced the results. Cyclomatic complexity, number of lines of code, and quantity of method invocations may not be sufficient to characterize the code as problematic. Therefore, the use of other code metrics could be useful to further explore the relationship between asserts and production code quality.

### C. External Validity

This study involved a few open source projects that belong to very different domains, use different architectural styles and design techniques, and have different sizes. However, we only studied projects from the Apache group, which follow some well and pre-defined code policies. Also, some of them receive more attention from the community than others. Therefore, this may have imposed limitations to the generalization of our conclusions.

Also, the number of selected projects is small. In this study, the number of projects was drastically reduced for

mainly two reasons:

1) Many projects were eliminated because they presented a small number of tests in comparison to their number of methods.
2) Many unit tests inside the Apache repository are, in fact, integration tests. These kind of tests usually make use of helper methods that belong to a different class. These methods are also responsible for asserting the output – many times, these outputs are not even classes; they are files, packets in a socket, and so on. This variation is not captured by the implemented tool. Therefore, we preferred to discard such projects. As the filter does not random the data, the results can not be generalized.

Finally, the number of industrial projects was too small, and it might be necessary to run the study over different industrial projects and check whether the results match those we obtained in this study.

## IX. Related Work

A study from Bruntink and van Deursen [8] discussed the reasons some classes are easier to be tested than others. In their work they related object-oriented metrics to test suite metrics. The OO metrics chosen was the Binder suite [7], which is highly based on Chidamber and Kemerer object-oriented metrics [10]. The test metrics were lines of code for a class, number of test cases, and number of assert instructions in a unit test. They were able to demonstrate a significant correlation between the class metrics (mainly in Fan Out, Size of the Class, and Response for a Class) and the test metrics (Size of the Class, and Number of Asserts). Interestingly, in our study, no correlation was found between asserts and production code metrics.

The work from Elish and Alshayeb [11] investigates the effects of refactoring in software testing effort. They showed that “Encapsulate Field,” “Extract Method,” and “Consolidate Conditional Expression” refactoring patterns increased the testing effort, while “Extract Class” reduced the testing effort. Additionally, “Hide Method” had no effect on the testing effort at all.

Heitlager *et al.* [16] discussed a maintainability model that relates high level maintainability characteristics to code level measures. The authors found that the complexity of source code had a negative correlation to the testability of the system. In other words, it means that the testability of the system tends to reduce as the complexity of source code increases.

The two last studies investigated the “other side” of the relationship: the effects of production code on the testing code. This reinforces the idea that both kinds of code are intrinsically connected and thus one influences the other. Such point can also be supported by studies in the TDD field that show an increase in code quality when developers write the test first [21] [22] [19] [27].

A study from Janzen [19] separated academic students in two different teams that employed opposite testing strategies, namely the test-first team and the test-last team. The study revealed that the code produced by the test-first team exhibited a better use of object-oriented concepts, as responsibilities were more adequately assigned to classes. In fact, the test-last team produced a more procedural kind of code. Furthermore, tested classes had 104% lower coupling measures than untested classes, and tested methods were 43% on average less complex than the untested ones.

Qualitatively, Williams *et al.* [15] noticed that 92% of participants in their study believed that TDD helped them achieve a higher quality code, and 79% believed that TDD promoted a simpler design.

## X. Conclusions and Future Work

In this paper, we studied the relationship between the quantity of asserts in a unit test and the quality of the associated production method being tested. Interestingly, related work also tried to observe a relation between specific aspects of a unit test and the quality of production code. To the best of our knowledge, our study is the first to relate not only the quantity of asserts, but also the number of instances that are asserted in unit tests to production code quality.

The main evidence we found is that when a production method contains associated unit tests that make asserts to more than one object instance, it tends to have more lines of code, a higher cyclomatic complexity, or make more method invocations than usual. This idea can benefit developers in general: when they feel the urge of asserting more than one object in a single test, it may be because the method being tested is more complex than it should be. Therefore, our answer to the question raised in the title of this paper is **the number of asserts does not tell us anything about the production code quality; however, the number of asserted objects does**.

As with any code smell, it does not necessarily point to a bad piece of code. Indeed, it is a cheap way to possibly find them. Developers then should pay attention to the suggested "More than One Asserted Object per Test" code smell, which can point out to methods with a higher cyclomatic complexity, number of lines of code, or even a higher number of method invocations.

A next step in this study would be to reduce the number of non-identified tests. In a future version, the tool should recognize asserts made in inherited or private methods. Also, separating the asserted objects by class type (to check whether the unit test validates the same class or a different one) may be interesting. Also, a qualitative study with the developers would help us understand why they wrote more than one assert per test.

It is important to highlight the fact that all industrial projects were not eliminated by the filter. Thus, running the

study in more industrial projects may add interesting data and improve external validity.

Finally, there might be other test aspects that could warn developers about code or design issues. Further evaluating the relationship between tests and code quality seems necessary.

## ACKNOWLEDGMENTS

The authors would like to thank Miguel Ferreira, Sandro Schulze, and Guilherme Travassos for their insightful ideas, suggestions, and reviews to the design and execution of this study. We also thank Caelum Learning and Innovation for supporting this research and allowing us to use their projects. Gustavo Oliva receives individual grant from the European Commission (EC) for his participation in the CHOReOS research project (www.choreos.eu). Marco Gerosa receives individual grant from CNPq.

## REFERENCES

[1] Kent Beck and Mike Beedle and Arie van Bennekum and, Alistair Cockburn and Ward Cunningham and Martin Fowler and James Grenning and Jim Highsmith and Andrew Hunt and Ron Jeffries and Jon Kern and Brian Marick and Robert C. Martin and Steve Mellor and Ken Schwaber and Jeff Sutherland Dave Thomas. Manifesto for Agile Software Development. http://agilemanifesto.org/. Last Access on September, the 27th, 2012. 2001.

[2] Aniche, M. F., Gerosa, M. A. How the Practice of TDD Influences Class Design in Object-Oriented Systems: Patterns of Unit Tests Feedback. Brazilian Symposium on Software Engineering, 2012.

[3] Aniche, M. F. Test-Driven Development: Teste e Design no Mundo Real. Casa do Código, 2012.

[4] Beck, K. Test Driven Development: By Example. Addison-Wesley Professional; 1st edition, 2002.

[5] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 1999.

[6] Beck, K. Aim, Fire. IEEE Software, Vol. 18, PP 87-89, 2011.

[7] Binder, R., Design for testability in object-oriented systems, Communications of the ACM 37, pp 87–101, 1994.

[8] Bruntink, M.; van Deursen, A. An empirical study into class testability. Journal of Systems and Software, vol. 79, pp. 1219-1232, 2006.

[9] Burbeck, S. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1987.

[10] Chidamber, S., Kemerer, C., A metrics suite for object oriented design, IEEE TSE, Vol. 20 (6), pp 476–493, 1994.

[11] Elish, K.O.; Alshayeb, M. Investigating the Effect of Refactoring on Software Testing Effort. APSEC '09, 2009.

[12] Feathers, M. The Deep Synergy Between Testability and Good Design. http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html, 2007. Last access on Oct 27, 2012.

[13] Fowler, M. Beck, K., et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1st Edition, 1999.

[14] Freeman, S.; Pryce, N. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional, 1st Ed., 2009.

[15] George, B.; Williams, L. An initial investigation of test driven development in industry. Proceedings of the 2003 ACM symposium on Applied computing. SAC '03, ACM, New York, NY, USA, 2003.

[16] Heitlager, I., Kuipers, T., Visser, J. A practical model for measuring maintainability. In QUATIC, pages 30–39, 2007.

[17] Henderson-Sellers, B., Object-oriented metrics : measures of complexity, Prentice-Hall, pp.142-147, 1996.

[18] Hunt, A.; Thomas, D. Pragmatic Unit Testing in Java with JUnit. The Pragmatic Programmers, 1st Edition, 2003.

[19] Janzen, D., Saiedian, H., On the Influence of Test- Driven Development on Software Design. Proceedings of the 19th CSEET, 2006.

[20] Janzen, D.; Saiedian, H.; Test-driven development concepts, taxonomy, and future direction. Computer, Volume 38, Issue 9, PP 43-50, 2005.

[21] Janzen, D., Software Architecture Improvement through Test-Driven Development. Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2005.

[22] Langr, J., Evolution of Test and Code Via Test-First Design, 02.12.2010, http://www.objectmentor.com/resources/articles/tfd.pdf

[23] Lehman, M., Perry, Dewayne, et al. Metrics and Laws of Software Evolution–The Nineties View., PP 20-32, 1997. Proceedings IEEE International Software Metrics Symposium (METRICS'97)

[24] Lorenz, M.; Kidd, J. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994.

[25] McCabe, T. A complexity Measure. IEEE TSE, SE-2, Vol. 4, pp 308-320, 1976.

[26] McGregor, John D.; Sykes, David A. A practical guide to testing object-oriented software. Addison-Wesley, 2001.

[27] Steinberg, D. H. The Effect of Unit Tests on Entry Points, Coupling and Cohesion in an Introductory Java Programming Course. XP Universe, Raleigh, North Carolina, USA, 2001.

[28] The Apache Software Foundation. http://www.apache.org/. Last access on 18th, January, 2012.

[29] van Deursen, Arie; Moonen, Leon; van den Bergh, Alex; Kok, Gerard. Refactoring Test Code. http://sqa.fyicenter.com/art/Refactoring-Test-Code.html. Last access on 9th, October, 2012.

[30] van Emden, E. Java quality assurance by detecting code smells. Proceedings of Ninth Working Conference on Reverse Engineering, 2002.

[31] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," J. Systems and Software, vol.23,no.2,1993.

[32] xUnit Patterns. Assertion Roulette. http://xunitpatterns.com/Assertion%20Roulette.html. Last access on Oct. 9th, 2012.