The Quality of Junit Tests

An Empirical Study Report

Dor D. Ma'ayan
Department of Computer Science
The Technion
Haifa, Israel
dorma10@campus.technion.ac.il

ABSTRACT

The quality of unit tests gains substantial importance in modern software systems. This work explores the way in which Junit tests are written in real world Java systems. We analyse 112 Java repositories and measure the quality of unit tests by finding patterns which indicate good practices of coding. Our results show that the quality of real world unit tests is low, and that in many cases, unit tests don't follow the well-known recommendations for writing unit tests. These early results demonstrate the need for more tools and techniques for refactoring of tests.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Software Engineering, Testing, Algebraic Representation, Taxonomy, Software Metrics

ACM Reference Format:
Dor D. Ma'ayan. 2018. The Quality of Junit Tests: An Empirical Study Report. In *SQUADE'18:: SQUADE'18:IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies , May 28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3194095.3194102

1 INTRODUCTION

Unit tests are taking a central part in modern code; they allow individual checking of properties in the program with complete isolation, and can be helpful for finding bugs during the development process. Automated testing frameworks, such as the *xUnit* collection transformed the task of writing unit tests to be easy and convenient then ever. However, the task of writing high quality unit test is still not trivial [3]. In addition, tests require maintenance as any other software component. Existing research explores ways to enhance the process of writing unit tests and maintaining them by defining metrics and by proving correlation between test attributes to the quality of tests in practise. There are also many blog posts, stack overflow issues, books and papers which discuss how to write quality unit tests [8]. However, a little is known on how do developers write unit tests in practise and whether developers use all these good practices while writing tests. Answering this question might reveal existing problems in tests design, both in terms of test structure and in API usability.

This short paper presents early results trying to address these questions. We analyse the Qualitas corpus containing 128 Java [2] repositories and detect patterns of Junit. Our results show that at least with Junit, there is a significant gap between recommendations to practise: 61% of all tests contain more than a single assertion, nearly 78% of all `assertTrue` and `assertFalse` assertions do not contain a customized error message and there is a poor usage of Hamcrest testing framework with less than 1% within all test methods using it.

Outline. *The reminder of this paper is organized as follows. A short description of well-known recommendations for writing good unit tests followed by our research questions is provided at Sect. 2, Sect. 3 describes the methodology of our empirical study.* . Sect. 4 presents preliminary results and answers the research questions, and Sect. 7 concludes.

2 RESEARCH QUESTIONS

The following section is organized as follows. Each subsection lists properties and good practices for writing unit tests and Junit in particular. Following each of these recommendations we present a research question which will be answered at Sect. 4.

2.1 The Single Checkpoint Principle

An important principle of any unit test is the *single checkpoint principle*: tests should have only one reason to fail and only one location for that failure to occur. In other words, each unit test should contain only one assertion, i.e., checking point. The *single checkpoint principle* is a solution to the Obscure Tests Problem [8, 10].

As an example, consider `t1()` Junit test at Fig. 1. This is an example of a classic Junit method with multiple assertions.

In a case of failure of that test, the developer does not know if a single correction will fix the test since there are too many check points with possible failure locations. In case where an assertion is failed, all the next assertions within the scope of the method will not get verified and the programmer will get a narrow picture over the correctness of the system. Possible solution is to verify that each test contains only one assertion.

Notice that in fact, it is not necessary that a method will contain a single assertion in order to follow the single checkpoint principle

```
public class T{
  private A a;
  private B b;
  private static final C C0 = new C();
  private static final C C1 = new C();
  //...
  @Test public void t1(){
    f(a);
    b = new B(a);
    final TM.T t = b.g(C0);
    assertNotNull(t);
    assertThat(t.a(), is(C1));
    assertThat(t.c(), is(C2));
    assertThat(t.i(), is(C3));
    assertThat(t.n(), is(C4));
    assertThat(t.g(), is(C5));
    assertThat(t.o(), is(C6));
  }

  @Test public void t2(){
  if(A.x == A.y)
    assertTrue(true);
  else
    assertTrue(false);
  }
  // More @Test methods
}
```

Figure 1: Junit `@Test` methods with multiple checking points i.e. multiple assertions

concept, for example, consider method `t2()` at Fig. 1 which is a decomposition of a single assertion. The importance of this principle rises the following research question:

RQ1: What percentage of Junit tests follow the single check point principle?

2.2 Test Structure

A common logical structure of unit tests follows the scheme [6] of:

$$Arrange \rightarrow Act \rightarrow Assert$$

Arrange all the necessary preconditions and test inputs,
Act by preforming a manipulation over a method or an object,
Assert that the expected result occurred.

This scheme of writing tests clearly separates what is being tested from the setup and verification steps, and also helps keeping the single checkpoint principle by making preforming of too many assertions at once to be harder. However, Repeating the arrange, act, assert scheme more than one time in a scope of a method violates the single checkpoint principle.

Another property of tests is their linearity, i.e, whether or not they contain complex flow control structures such as loops, condition statements, return statements ect. Keeping tests linear produce more readable and maintainable tests.

RQ2: What percentage of Junit tests follow the arrange, act, assert scheme? What are the common structures for Junit tests?

2.3 Using the Right Assertion

Using appropriate assertions proved to have strong correlation with the test suite effectiveness [12]. Junit offers more than ten different kinds of assertions for various common purposes. Using the wrong assertion and avoiding error messages harms the comprehensibility of the tests suite. A solution might be adding assertion messages, as well as making sure to use the most suitable assertion for the test. An example of using the wrong assertion might be: `assertTrue(a==b)` instead of `assertEquals(a,b)`: using the latest assertion produce more readable code and also produce more relevant error message. In fact, the usage of `assertTrue` and `assertFalse` produce the same error message as of the regular Java assertion, which in practise does not provide any meaningful information.

Appropriate assertions makes the test code more readable and maintainable. Hamcrest, which is relatively new extension of Junit provides fluent-api style assertions which are easier to follow by programmers.

RQ3: What is the distribution of usage of different kinds of Junit assertions?

3 RESEARCH METHODOLOGY

We introduce a static analysis technique for analysing of large corpuses of code. For each corpus, we traverse over all projects within it, and in each project traverse over all its Java files. Each Java file is translated into an AST, and during a traversal over the AST, patterns are recognized. Finally, all the data is stored in a CSV file for a further analysis.

The AST analysis is based on work done by the author and other over the Spartanizer and Athenizer projects [4]. An overview of our analysis methodology is described using Fig. 2.

Currently, our analysis supports Junit tests [1]. Junit is the most popular Java testing framework, and studies show that nearly 97% of the Java projects use Junit at a certain point during their lifetime [11].

4 RESULTS

The research methodology has been applied over the Qualitas corpus [9] containing 112 Java systems[1] [2]. After filtering of other testing frameworks, we found that 70 out the 112 projects use Junit as their testing framework. This is a clear indication for the popularity of Junit. In total, we analysed 140,146 Java files with 55,848 Junit tests.

The rest of this section provides answers to the research questions based on the Qualitas analysis.

RQ1: What percentage of Junit tests follow the single check point principle?

Fig. 3 shows a diagram of the number of test methods categorized by the number of assertions within them.

As can be seen, 61% of the testing methods detected violate the single checkpoint principle, meaning they contain more than a single assertion per method. In addition, 18% of the methods contain six assertions or more. This results indicate that many tests tend to be long and complex and in fact, violate the isolation principle that unit tests provide.

[1] A full description of the code can be found at http://qualitascorpus.com/docs/catalogue/20130901/index.html
[2] We used version 20130901 of Qualitas

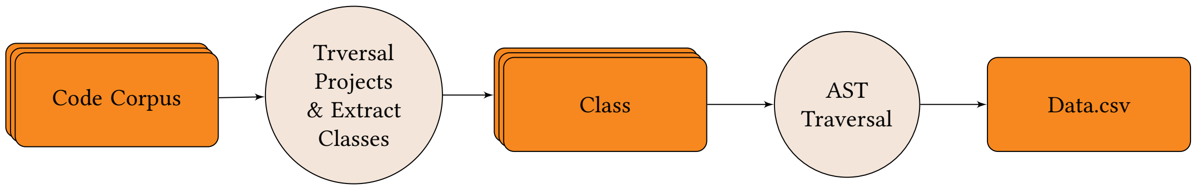

Figure 2: Methodology Overview

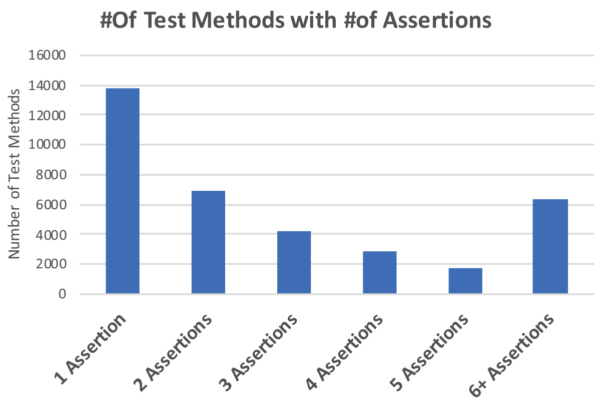

Figure 3: Diagram of the number of test methods categorized by the number of assertions within them.

For checking methods with multiple assertion who follow the single checkpoint principle, we manually checked 1,000 testing methods from the corpus with more than one assertion and found non of them to follow the single checkpoint principle.

RQ2: What percentage of Junit tests follow the arrange, act, assert scheme? What are the common structures for Junit tests?

For well defined exploration of test structures, we introduce a taxonomy for test structures which is based on regular expressions, for example, consider the following expressions over group X:

- **X** Represents exactly one occurrence of items from X.
- **X*** Represents at least zero occurrences of items of X.
- **X**$^+$ Represents at least one or more occurrences of items of X.

We define those regular expressions over the following 2 groups:

- **F** the fixtures group, contains all the commands within tests which are not assertions, and
- **A** the assertions group, contains all the assertion command within a test.

For example, representing the arrange, act, assert scheme using our taxonomy is done using:

$$F^*A^*$$

and for representing the single checkpoint principle is done using:

$$F^*A$$

This taxonomy represents only linear tests. Linear tests are test methods which do not contain any complex control structure within them, but instead, a sequence of commands and assertions which are executed in sequential order. As mentioned, linear tests are more readable and maintainable. As a first stage, we filtered all the non-linear tests and found that nearly 31% of the entire tests are non-linear. Table 1 shows the prevalence of different structure of tests within the Qualitas corpus.

Table 1: Taxonomy of tests and their prevalence in Qualitas

Algebraic Form.	Qualitas
A	2.6%
F	12.7%
A^+	5.2%
F^+	25.9%
FA	3.3%
F^+A	13.2%
FA^+	5.3%
F^+A^+	22.1%
Non Linear	30.7%

Note that the percentages in the table are calculated from the total number of tests, including the non-linear tests. Also, note that there are few patterns which are sub-patterns of others, for example, F is a sub-pattern of F^+.

A significant percentage of methods match the pattern F^+, which means that in practise, they do not contain any Junit assertion. The reason is that apparently, there are many test methods which use the general framework of Junit excluding the Junit assertion and instead, use their own customized assertions. The current methodology doesn't recognize such methods and they require a further research in the future.

Only 13% of the methods follow the arrange, act assert scheme, and more than 12% of the methods (excluding the non linear methods) contain a mixture of fixtures and assertions which might lead to low readability and maintainability of these methods.

As mentioned, 31% of the tests are non-linear, from those tests, we found that 13% contains loops (`for` statements and `while` statements), 16.5% contains `try-catch` statements (Although Junit allows declare throwed exception for each testing methods), and 15% contains `if` statements.

RQ3: What is the distribution of usage of different kinds of Junit assertions?

Fig. 4 shows a graph of the distribution of Junit assertion within the Qualitas corpus, grouped by different kinds of assertions:

The most popular assertion in Qualitas is `assertEquals` which takes nearly 50% of the total assertions.

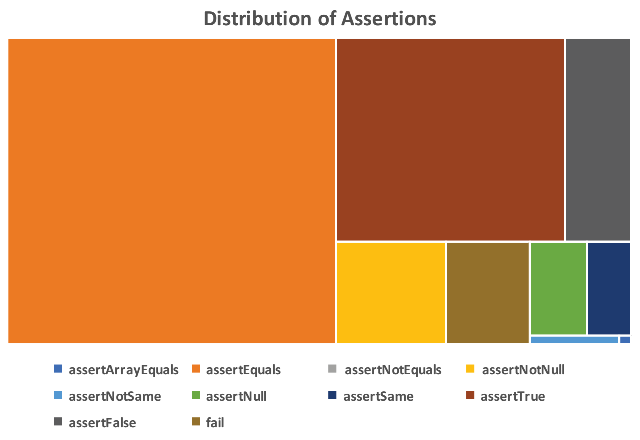

Figure 4: Graph of the distribution of Junit Assertions in the Qualitas corpus, grouped by different kinds of assertions

As mentioned, the Hamcrest framework provides better assertion messages using a fluent api style. It uses `assertThat` assertion containing a matcher. However, less then 0.3% of the assertions are Hamcrest assertions. Another popular assertion is `assertTrue`. Using `assertTrue` and `assertFalse` without a user defined error message is considered a bad practise since the default error message of the test does not provide any indication for the reason of failure. Out of all `assertTrue` and `assertFalse` assertions in Qualitas, 77% doesn't contain any customized error message. This indication by itself gives a strong motivation for auto generation of error messages with dedicated tools.

5 THREATS TO VALIDITY

The main threat to the validity of our results is the corpus. We used the latest version of the Qualitas corpus, which is dated back to 2013. New frameworks and automation techniques might have been introduced since then and affect the results. For example, we assume that the Hamcrest framework gained significant popularity since 2013.

This paper describes preliminary results on the usage of Junit, any other testing framework was filtered out. In addition, we have found that many systems use the general framework of Junit with their own assertion methods which are more domain-specific. Considering these user defined assertions in the analysis might change the results in an unpredictable way.

6 RELATED WORK

Empirical study of tests has increasing interest lately with multiple works address similar questions. Gonzalez et.al. [5] preformed a large scale empirical research over 4 testing patterns and found that only 24% of the projects implement pattern that could help with maintainability. Zerouali et.al. [11] explored the usage of different testing framework. Kochar et.al. also conducted a large-scale study on tests [7] using 20,817 GitHub projects. They studied the correlations between different statistics such as the number of contributors, the project size, and also the relationship between programming languages to the number of test cases in a project. Our work looks with finer granularity over test methods in terms of structure and API usability.

7 CONCLUSION

This paper explores how Junit tests are written in real world software systems. We preformed a static analysis over the Qualitas corpus and found that significant part of the testing methods do not follow the recommended practices for writing unit tests. These early results might demonstrate the importance of automation tools and the need for refactoring techniques for unit tests. As a future work, we plan to reproduce our results over a bigger and more up-to-date Java corpus, extend the research for more testing frameworks, and trying to find correlations between our results to traditional testing metrics such as code coverage by tests.

Acknowledgements *I thank my advisor, Prof. Joseph Gil from the Technion for his professional guidance and help. I also thank my colleague, Ori Roth for his help reviewing this work and making it better. The financial support of the Technion is also appreciated.*

REFERENCES

[1] Sujoy Acharya. 2014. *Mastering Unit Testing Using Mockito and JUnit.* Packt Publishing Ltd, Birmingham, UK.
[2] Ken Arnold and James Gosling. 1996. *The Java Programming Language.* Addison-Wesley Publishing Company, Reading, MA.
[3] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/2786805.2786843
[4] Yossi Gil, Dor Ma'ayan, Niv Shalmon, Raviv Rachmiel, and Ori Roth. 2017. Syntactic Zoom-Out/Zoom-In Code with the Athenizer. In *Software Visualization (VISSOFT), 2017 IEEE Working Conference on.* IEEE, IEEE, Piscataway, NJ, USA, 124–128.
[5] Danielle Gonzalez, Joanna C. S. Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A Large-scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17).* IEEE Press, Piscataway, NJ, USA, Article 1, 11 pages. https://doi.org/10.1109/MSR.2017.8
[6] Jeff Grigg. 2012. Arrange Act Assert. http://wiki.c2.com/?ArrangeActAssert. (2012).
[7] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. An empirical study of adoption of software testing in open source projects. In *Quality Software (QSIC), 2013 13th International Conference on.* IEEE, IEEE, Piscataway, NJ, USA, 103–112.
[8] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education, London, UK.
[9] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010).* IEEE, Piscataway, NJ, USA, 336–345. https://doi.org/10.1109/APSEC.2010.46
[10] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001).* ACM, New York, NY, 92–95.
[11] Ahmed Zerouali and Tom Mens. 2017. Analyzing the evolution of testing library usage in open source Java projects. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on.* IEEE, IEEE, Piscataway, NJ, USA, 417–421.
[12] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/2786805.2786858