Attention, Test Code is Low-quality!

Xinye Tang
State Key Laboratory of Computer Science
Institute of Software,
Chinese Academy of Sciences
tangxinye@nfs.iscas.ac.cn

ABSTRACT
In this paper, we describe the formatting guidelines for ACM SIG Proceedings. Software testing is an essential process during software development and maintenance for improving software quality. Test code, the artefact during software testing, has been widely used in many software quality assurance techniques.

Traditionally, software quality assurance techniques, e.g., automatic bug repair, fault localization, test case prioritization, and mining API usage from test code are based on the hypothesis of a sound quality of the test code. However, via empirical study on four open source projects, we found that the quality of test code is quite low comparing with corresponding source code, and this might hurt the above software quality assurance techniques.

In this paper, we studied more than 140,000 LOC(lines of code) test code from four large scale and widely used open source projects and found that it is common for test code to be unregulated and of low-quality in open source projects. First, the comment clone ratio, unreleased resource ratio and clone code ratio of test code is much higher than that of corresponding source code; second, bug-fixed coverage is down to 0. We have learned the following lessons: the quality of test code is quite low comparing with corresponding source code, and the low quality test code may misguide existing software quality assurance techniques.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: [Testing and Debugging]

General Terms
Experimentation, Measurement.

Keywords
Test code quality, empirical study, testing, software quality assurance.

1. INTRODUCTION
Software testing is an essential process during software development and maintenance. Test code is widely used as the artefact during software testing for ensuring software quality. Therefore, it's critical to maintain high quality test code. Existing work [3] revealed that high quality test code of a software system could improve the development team's performance. Moreover, they reported that the bad quality of test code demonstrated a significant positive correlation between test code quality and the throughput and productivity of issue handling. In this paper, we conduct a pilot study to examine test code quality from a different aspect, specifically, we try to study the potential relation between test code quality and some software quality assurance techniques, e.g., automatic bug repair, fault localization, test case prioritization, and mining API usage from test code.

Traditionally, software quality assurance techniques, e.g., automatic bug repair [4, 9, 14], fault localization [11, 15], test case prioritization [6], and mining APIs from test code [8][16] are based on the hypothesis of a sound quality of the test code. Research studies on automatic fault repair leverage test code to measure their performance. For fault localization, test code is required to improve the accuracy and establish the lower and upper bounds. Test case Prioritization techniques aim to rearrange the execution order of test cases, which is based on the sound quality of the test code. What's more, test code is essential to mine API usage examples, which is helpful for developers to learn and understand the correct usage of APIs of libraries.

In this paper, via empirical study on four large scale and widely used open source projects, we found that the quality of test code is quite low comparing with corresponding source code, and this might have negative impact on the above software quality assurance techniques. We studied more than 140,000 LOC test code from four large scale and widely used open source projects and found that it is common for test code to be unregulated and low-quality in open source projects. Specifically, first the comment clone ratio, unreleased resource ratio and clone code ratio of test code is much higher than that of corresponding source code; second, bug-fixed coverage is down to 0.

We have learned the following lessons: the quality of test code is quite low comparing with corresponding source code, and the low quality test code may misguide existing software quality assurance techniques. The main contributions of this work include:

1. We proposed five criteria for the measurement of test code quality.

2. Based on proposed criteria, we measured test code quality of four large scale, widely used open source projects. Results show that the quality of test code is quite low comparing with corresponding source code. We further discuss the potential

Copyright © 2015 for this paper by its authors. Copying permitted for private and academic purposes

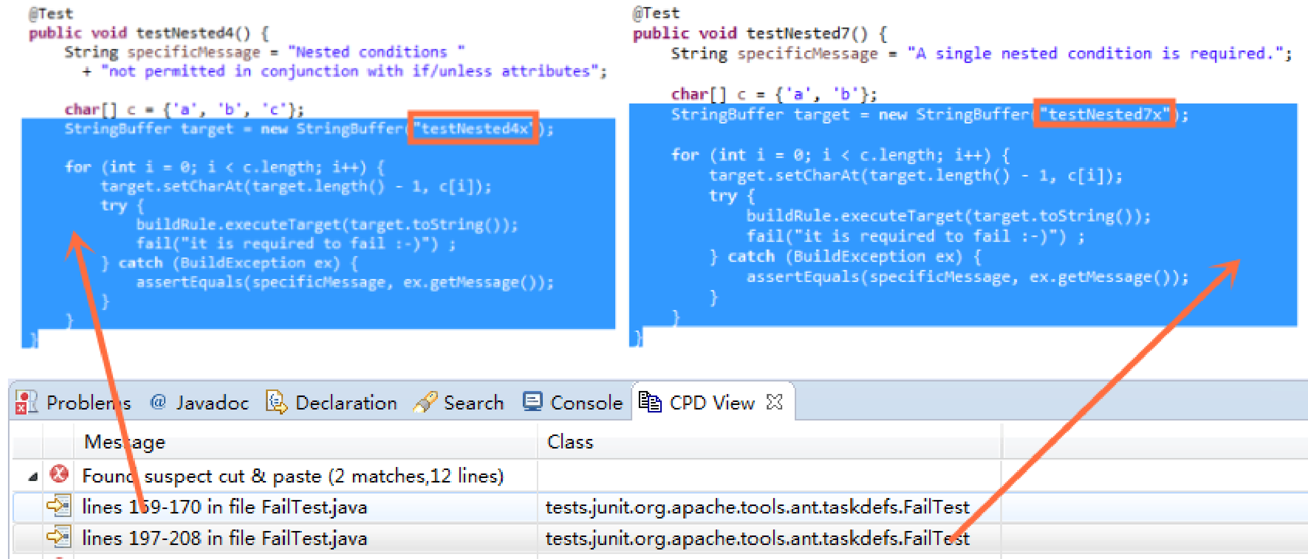

Figure 1: An example of duplicate code extracted from FailTest.java in Ant 1.9.4.

impact of low quality test code on existing software quality assurance techniques. To the best of our knowledge, this is the first work to report that the test code is low-quality and untrustworthy, which should be taken seriously.

In the remainder of this paper, section II presents points on which the author would like to get the most advice on; Section III presents essential background and related work of our study; Section IV shows our motivation; Section V present the three categories for the measurement of the test code quality. Section VI explains how we conduct our empirical study; Section VII discusses the threats to this work; Section VIII concludes this paper and discuss our future work.

2. ADVICE WANTED

As for the points on which we would like to get the most advice on, we are thinking about the possibilities of proceeding our research further. Specifically, we plan to conduct quantitative and qualitative studies to explore how exactly the low-quality test code could impact software quality assurance techniques, i.e., automatic bug repair, fault localization, test case prioritization, and mining API usage. We will appreciate it if mentors could give insightful suggestions on whether the work is valuable and how it could be effectively done. Also, any feedback on the structure and content of the paper would be welcome.

3. BACKGROUND AND RELATED WORK

Automatic Bug Repair: is the process of automatically generating patches for repairing bugs. A lot of studies have been carried out to address this issue. Weimer et al. [14] present a fully automated technique for repairing bugs using one part of a program as a template to repair another part. Kim et al. [9] proposed a patch generation approach learned from human-written patches and identified common fix patterns for automatic patch generation. Tan et al. [13] propose an approach of automated repair of software regression bugs. Test code is used to evaluate the effectiveness of these approaches, specifically, given a bug, if a generated repair patch could pass all test cases, the generated repair patch will be treated as an effective repair for this bug.

Fault Localization: is the indispensable process to identify exactly where the bugs are before fixing them. Xuan et al. [15] pointed out that the effectiveness of fault localization depends on the quantity of test code and proposed an approach to improve fault localization with test case purification. Steimann et al. [11] empirically explored performance of existing fault locators and their results shown the quality of test code is a key factor for fault locators. Campos et al. [5] proposed an approach to fault localization by entropy-based test generation.

Test Case Prioritization: aims to rearrange the execution order of test cases to maximize specific objectives. Elbaum et al. [6] compare different test case prioritization techniques in regression testing on the performance in improving the rate of fault detection.

Mining API Usage from Test Code: Understanding and learning the correct usage of APIs of libraries are significant but complex activities for developers. Ghafari et al. [8] described an approach to code recommendation where examples are obtained by mining and manipulating the unit tests of the API. Zhu et al. [16] proposed an approach to mining API usage examples from test code, combining the technique of clustering to improve the representativeness of extracted examples. Nasehi et al. [10] proposed to supplement the standard API documentation with relevant examples taken from the unit tests. Thus, the quality of APIs usage in test code is critical to this topic.

Test Code Quality: It is critical to detect the quality of test code, however there are not enough work done in this field. Athanasiou et al. [3] revealed that high quality test code of a

Table 1. Details of Studied Projects in This Work

Project	Version		TC*	Clone Com#	Clone Com Ratio#	Unreleased Resources#	Unreleased Resource Ratio#	Clone Codes#	LOC#	Clone Code Ratio#
Ant	1.9.4	Test	329	35	0.11	67	0.20	6863	43774	0.16
		Src	832	25	0.03	101	0.12	15918	183692	0.09
Maven	3.2.5	Test	175	29	0.17	35	0.2	2816	18581	0.15
		Src	659	18	0.03	27	0.04	5801	77983	0.07
Log4j	1.2.17	Test	90	26	0.29	29	0.32	2993	13681	0.22
		Src	272	28	0.10	33	0.12	4757	41991	0.11
Commons Math	3.5	Test	570	114	0.2	41	0.07	38986	98626	0.40
		Src	942	242	0.26	19	0.02	29295	95697	0.31

* TC represents Total Classes, which means the number of the test classes.LOC represents lines of code.

Clone Com(Comment) are the number of test classes which clone comment, and Clone Comment Ratio represents the number of clone comment classes out of the total number of test classes. Unreleased Resources are the number of test classes which do not release resources, and Unreleased Resource Ratio represents the number of test classes which do not release resource out of the total number of test classes. Code Clones represent the lines of clone code, and Clone Code Ratio represents the lines of clone code out of the total lines of code.

Software system could improve the development team's performance.

4. MOTIVATION

As is mentioned above, test code is closely related with software quality assurance techniques, e.g., automatic bug repair, fault localization, test case prioritization, and mining API usage from test code. These studies are based on the hypothesis of a sound quality of the test code. Thus, the quality of test code is critical for the performance of these techniques. In this study, we try to conduct a pilot study to explore the quality of test code according to the five criteria.

5. TEST CODE QUALITY

In this section, we present the three categories for the measurement of test code quality.

5.1 Incorrectness

This category consists of test criteria that focus on measuring the error detection ability of the code. Unreleased resource and code clone are the main criteria for this category.

5.1.1 Unreleased resource

Unreleased resource is a kind of incorrect use of APIs. Unreleased resource occurs when developers fail to release resource such as File, ResultScanner and so on. When developers finished the input and output operations on a file object, developers should close file and release resources, or there will be a potential memory leak vulnerability. Moreover, developers should close the resource with finally clause to ensure the resource is closed no matter what happens in the try block.

The following real test case from Ant 1.9.4, is an example where the method did not close the file object it opened.

```
1 public void testPassFile () throws Exception {
2    buildRule . executeTarget (”test3 ”);
3    File f = new File (
4       buildRule . getProject (). getBaseDir () ,
5       “testpassfile .tmp ”);
6    assertTrue ( . . . ) ;
7    assertEquals ( . . . ) ;
8 }
```

In this study, the unreleased resource ratio is defined as the following:

Unreleased Resource Ratio = (#unreleased resource classes)/TC

where ***#unreleased resource classes*** shows the number of the classes which did not release resources, and ***TC*** shows the total number of the classes.

We have developed a tool to automatically examine the code, and checked whether every resource was closed after being opened and used. If the resource was closed, we checked whether the close statement is enclosed in finally blocks. We counted both the resources which were not closed and those that were not closed in finally block for the unreleased resources.

5.1.2 Code Clones

Code clones are separate fragments of code that are very similar. They are a common phenomenon in the open source systems which have been under development for some time. Clone codes are often referred to due to the difficulty it makes in changing and maintaining the open source systems since developers have to locate and update many fragments frequently. For example, Fowler [7] argues that code duplicates are bad smells of poor

designFigure 1 shows an example of clone codes. Clone code ratio is defined as the following:

Clone Code Ratio = (#clone code lines)/LOC

where ***#clone code line*** shows the lines of clone code, and ***LOC*** shows the lines of code.

To estimate the clone code, we use PMD's Copy/Paste Detector (CPD) [2] and set the minimum Tile-size at 10, meaning that CPD cannot find clones in methods that are less than 10 statements long. Figure 1 indicates that the clone analysis tool can find fragments which differ in the names of variables and parameters, and in which some statements have been rearranged.

5.2 Insufficiency

Criteria that fall inside this category focus on measuring the loss detection ability of the code. To identify the loss of code, we choose code coverage and bug-fixed coverage as the main criteria.

5.2.1 Code Coverage

Code coverage is the most frequently used metric for test code quality assessment [3]. It is used to describe the degree to which the source code of a project is tested by a particular test suite. Usually the project with lower code coverage has been insufficiently tested and therefore has a higher chance of containing bugs. There exist many tools for dynamic code coverage estimation (e.g., Clover 5 and Cobertura 6 for Java, Testwell CTC++ 7 for C++, NCover 8 for C#). We used Clover [1] to obtain a code coverage metric.

5.2.2 Bug-fixed Coverage

In general, after fixing a bug, new unit test should be created and added to the regression test suite to ensure this bug will not be re-introduced in the following versions of the projects. The bug-fixed coverage is a measure used to describe the degree to which extent the fixed bug is covered by test code. If a fixed bug has not been covered by test cases, this fixed bug will be under high risk of reopening. In this study, the bug-fixed coverage is defined as the following:

Bug-fixed Coverage = (#tested bugs)/ (#fixed bugs)

where ***#tested bugs*** is the number of fixed bugs which is tested in the current version of the project, and ***#fixed bugs*** is the number of fixed bugs in the current version of the project.

To calculate the bug-fixed coverage, we first collected the fixed bugs for the responding version of the four open source projects from their release notes; second, we found the fixed bugs which are not tested in the current version, manually. Specifically, the first author is responsible for the collection of un-tested fixed bugs, after which the second author recollected the data. Then results are merged, and conflicts were get resolved by a joint pair-inspection of all three authors.

5.3 Bad Readability

This category consists of test criteria that focus on measuring the detection of the unreadable code. Clone comment ratio is the main criterion for this category.Comments make the code developer-readable. They generate code documentation in predefined format. Clone comments are the same prologue comment of different methods, which makes the code inconsistent with the code documentation and hard to understand. Often this occurs when developers miss to change the clone comment.

Consider the following real test code from Ant 1.9.4:

```
1 /** Test right use of cache names. */
2 @Test
3 public void testValidateWrongCache () {
4    . . .
5 }
6 /** Test right use of cache names. */
7 @Test
8 public void testValidateWrongAlgorithm() {
9    . . .
10 }
```

In this study, the clone comment ratio is defined as the following:

Clone Comment Ratio = (#clone comment classes)/TC

where ***#clone comment classes*** shows the number of the classes which clone comment, and ***TC*** shows the total number of the classes.

The logic of the two test cases are different, yet the comments are the same, which might cause the developers' confusion in terms of the understanding of the test logic of these two test cases. We developed a simple tool to detect cloned comments, which could look into the code and locate the functions that share comment, and collect the clone comments automatically.

6. EMPIRICAL STUDY

In this section, we present the quality of test code of the four open source projects according to the five criterion respectively.

6.1 Dataset

In this paper, we try to explore the quality of test code of four large-scale and widely used open source projects, i.e., Ant, Maven, Log4j, and Commons Math, using the five criteria. For each project, we extracted the test code and source code separately from the latest version. Details of these projects are shown in Table 1.

6.2 Result Analysis

6.2.1 Unreleased Resource

As is shown in Table 1, in the four projects, the unreleased resource ratio for test code is up to 32% and on average it is 20%, while the average ratio for source code is less than 7%. Overall, all the unreleased resource ratios in test code are much higher than that in the corresponding source code.

Potential Impact on Software Quality Assurance Techniques: The results indicate that the quality of source code is much higher than test code. As the test code is widely used for mining API usage examples [10, 16], this can result in bad API usage examples, making API learning quite confusing for developers.

Table 2. Details of Quality of Test Code

Project	Version	Code Coverage	Tested FBs#	FBs#	Bug-fixed Coverage#
Ant	1.9.4	85%	0	15	0.00
Maven	3.25	78%	1	6	16.67
Log4j	1.2.17	75%	0	5	0.00
Commons Math	3.5	81%	2	8	25.00

FBs are the number of fixed bugs in the current version, and Tested FBs are the number of fixed bugs which have been tested in the current version. Bug Coverage is the ratio of Test FBs out of FBs.

6.2.2 Code Clone

As is shown in Table 1, in the open source projects, the clone code ratio for test code is up to 40% and on average the ratio is about 23%, while the average ratio for source code is about 17%. Overall, all the clone code ratios in test code are much higher than that in corresponding source code.

Potential Impact on Software Quality Assurance Techniques: Clone test code is harmful for software quality, which increases test maintenance overhead and propagates any pre-existing errors. Clone test code always cover similar source code, which reduces the discrimination among test code and might hurt fault localization and test case prioritization.

6.2.3 Code Coverage

As is shown in Table 2, in the four projects, the code coverage for test code varies from 75% to 85%, while the average ratio for code coverage is around 80 %. Overall, the code coverage are high enough to basically cover the test of the source code.

6.2.4 Bug-fixed Coverage

Results are shown in Table 2, Bug-fixed Coverage in our studied versions of Ant and Log4j is 0, which means none of the fixed 20 bugs in Ant 1.9.4 and Log4j 1.2.17 has been tested after fixing. In Maven, and Commons Math, values of Bug-fixed Coverage are also quite low, more than 70% fixed bugs are not addressed in test code. Overall, the bug-fixed coverage is so low that the test for the fixed bugs are not sufficient.

Potential Impact on Software Quality Assurance Techniques: The low Bug-fixed Coverage values in software projects make it hard to practice test case prioritization, and also hurt the effectiveness of automatic bug repair. Since test case prioritization aims to arrange test cases based on the code coverage for accelerating bug detection, the low Bug-fixed Coverage means that when un-tested fixed bugs are re-introduced, prioritized test cases might not reveal these bugs; for automatic bug repair, the low Bug-fixed Coverage makes the evaluation inefficient. Automatic bug repair evaluates its repair patch by running all test cases. If all test cases are passed, the generated patch will be treated as an effective repair patch for a bug. However the low Bug-fixed Coverage means many fixed bugs are not tested by the existing test cases. So when evaluating generated repair patches, even all test cases are passed, these un-covered fixed bugs might be re-introduced.

6.2.5 Comment Clone

As is shown in Table 1, the average clone comment ratio for test code is about 20% and the average clone comment ratio for source code is less than 10%. Overall, the clone comment ratio in test code is much higher than that in source code of the four projects. The results indicate that the quality of source code is much higher than the test code in terms of comment clone.

Potential Impact on Software Quality Assurance Techniques: Program comments are important for developers to understand code. Moreover, comments that are inconsistent with code can easily confuse and misguide developers to introduce bugs in subsequent versions [12]. The high clone comment ratio in test code is harmful for developers to understand the test logic.

7. THREATS TO VALIDITY

7.1 Internal Validity

In this paper, we study the quality of test code. we present the five criteria for the measurement of test code quality. However, other criteria that we have overlooked may also can measure the quality of test code.

7.2 External Validity

In this work we investigate the quality of test code in terms of five proposed criteria on open source projects. However, it is possible that our approach may not work well on some closed-source software, or small scale open source software projects. The purpose of this work is to study the impact of low quality test code on several software assurance techniques, however, not all projects maintain valid test code. Our approach is not suitable for these projects without test code.

8. CONCLUSION AND FUTURE WORK

This paper found that it is common for test code to be unregulated and of low quality in the open source projects. We studied 1164 test classes and more than 140,000 LOC test code from the current version of four open source projects. Results indicate that the quality of test code is much lower than that for the corresponding source code in terms of the proposed criteria, e.g., unreleased resource, code clone and comment clone, and the the coverage of test code for fixed bug is not sufficient.

We further discussed the potential impact of low quality on the existing software quality assurance techniques. To the best of our knowledge, this is the first work to report that the test code is low-quality and untrustworthy, which should be taken seriously.

Future work. Our research is in a final stage. We explored the quality of test code in terms of five proposed criteria and the impact of test code on software quality assurance techniques. In the future, we plan to conduct quantitative and qualitative studies to explore how exactly the low-quality test code could impact

software quality assurance techniques, i.e., automatic bug repair, fault localization, test case prioritization, and mining API usage.

Acknowledgment

This research was supported in part by National Natural Science Foundation of China under Grant Nos. 91218302, 91318301, 71101138, and 61303163.

References

[1] Clover. https://www.atlassian.com/software/clover/overview. Accessed April 20, 2015.

[2] Pmd's cpd. http://pmd.sourceforge.net/pmd-4.3.0/cpd.html.

[3] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman.Test code quality and its relation to issue handling performance. volume 40, pages 1100–1125, Nov 2014.

[4] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014), Hong Kong, volume 16.

[5] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based test generation for improved fault localization. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 257–267.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. volume 28, pages 159–182, 2002.

[7] M. Fowler. Refactoring: Improving the design of existing code. In Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002, page 256, 2002.

[8] M. Ghafari, C. Ghezzi, A. Mocci, and G. Tamburrelli. Mining unit tests for code recommendation. In Proceedings of the 22Nd International Conference on Program Comprehension, pages 142–145. ACM, 2014.

[9] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In Proceedings of the 2013 International Conference on Software Engineering, pages 802–811. IEEE Press.

[10] S. M. Nasehi and F. Maurer. Unit tests as api usage examples. In Software Maintenance (ICSM), 2010 IEEE International Conference on, pages 1–10. IEEE, 2010.

[11] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, pages 314–324. ACM.

[12] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07), October 2007.

[13] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In Proceedings of the 2015 International Conference on Software Engineering. IEEE, 2015.

[14] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, pages 364–374. IEEE Computer Society, 2009.

[15] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 52–63. ACM, 2014.

[16] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang. Mining api usage examples from test code. In Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pages 301–310. IEEE.