Does Unit-Tested Code Crash? A Case Study of Eclipse

Efstathia Chioteli*
Ioannis Batas*
Diomidis Spinellis*
t8150148@aueb.gr
t8150090@aueb.gr
dds@aueb.gr
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece

ABSTRACT

Context: Software development projects increasingly adopt unit testing as a way to identify and correct program faults early in the construction process. Code that is unit tested should therefore have fewer failures associated with it.
Objective: Compare the number of field failures arising in unit tested code against those arising in code that has not been unit tested.
Method: We retrieved 2 083 979 crash incident reports associated with the Eclipse integrated development environment project, and processed them to obtain a set of 126 026 unique program failure stack traces associated with a specific popular release. We then run the JaCoCo code test coverage analysis on the same release, obtaining results on the line, instruction, and branch-level coverage of 216 392 methods. We also extracted from the source code the classes that are linked to a corresponding test class so as to limit test code coverage results to 1 263 classes with actual tests. Finally, we correlated unit tests with failures at the level of 9 523 failing tested methods.
Results: Unit-tested code does not appear to be associated with fewer failures.
Conclusion: Unit testing on its own may not be a sufficient method for preventing program failures.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

*Chioteli collected the unit testing results and performed the qualitative analysis. Batas performed the quantitative analysis. All authors contributed equally to the paper's writing.

PCI '21, November 26–28, 2021, Volos, Greece

ACM ISBN 978-1-4503-9555-7/21/11…$15.00
https://doi.org/10.1145/3503823.3503872

KEYWORDS

Unit-testing, crash incident reports, code coverage, stack traces, software reliability

ACM Reference Format:
Efstathia Chioteli, Ioannis Batas, and Diomidis Spinellis. 2021. Does Unit-Tested Code Crash? A Case Study of Eclipse. In *PCI '21: 25th Pan-Hellenic Conference on Informatics, November 26–28, 2021, Volos, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3503823.3503872

1 INTRODUCTION

The rising size and complexity of software multiply the demands put on adequate software testing [16]. Consequently, software development projects increasingly adopt unit testing [5] or even test-driven development [4] as a way to identify and correct program faults early in the construction process. However, the development of testing code does not come for free. Researchers have identified that one of the key reasons for the limited adoption of test-driven development is the increased development time [6]. It is therefore natural to wonder whether the investment in testing a program's code pays back through fewer faults or failures.

To examine how test code coverage relates to software quality, numerous methods can be employed. In this study we investigate the relationship between unit testing and failures by examining the usage of unit testing on code that is associated with failures in the field. We do this in three conceptual steps. First, we run software tests under code coverage analysis to determine which methods have been unit tested and to what extent. We triangulate these results with heuristics regarding the naming of classes for which unit test code actually exists. Then, we analyze the stack traces associated with software failure reports to determine which methods were associated with a specific failure. Finally, we combine the two result sets and analyze how unit-tested methods relate to observed failures.

We frame our investigation in this context through the following research questions.
RQ1 How does the testing of methods relate to observed failures?
RQ2 Why do unit-tested methods fail?

A finding of fewer failures associated with tested code would support the theory that unit testing is effective in improving software reliability. Failing to see such a relationship would mean that further research is required in the areas of unit test effectiveness

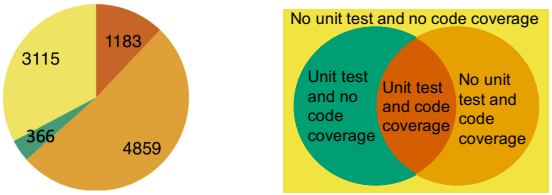

Figure 1: Relationship between test code coverage at the level methods and existence of unit tests at the level of classes. The pie areas correspond to the colored areas of the Venn diagram depicted on the right.

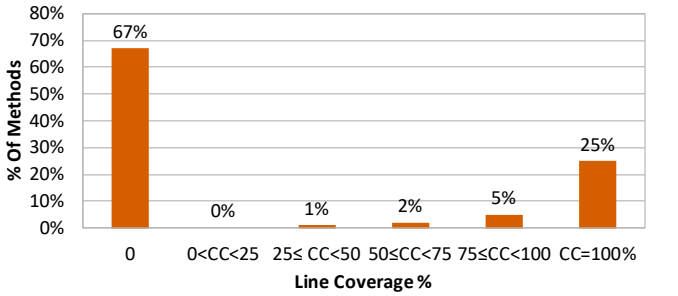

Figure 2: Relationship between code coverage and methods of JaCoCo report.

Table 1: Crashes of Tested Methods

	Unit tested					
Crashed	No		Yes		Total	
No	7816	(82.1%)	541	(5.7%)	8357	(87.8%)
Yes	1097	(11.5%)	69	(0.7%)	1166	(12.2%)
Total	8913	(93.6%)	610	(6.4%)	9523	

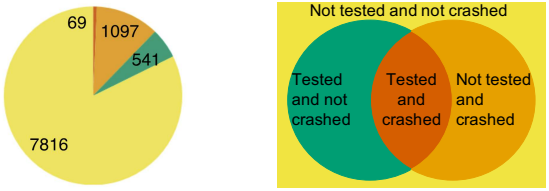

Figure 3: Relationship between strictly tested and strictly crashed methods. The pie areas correspond to the colored areas of the Venn diagram.

(why were specific faults not caught by unit tests) and test coverage analysis (how can coverage criteria be improved to expose untested faults).

The main contributions of our study are the following:

- a method for investigating the effectiveness of unit testing,
- an empirical evaluation between unit test coverage and failure reports, and
- an open science data set and replication package providing empirical backing and replicability for our findings.

2 METHOD

An overview of the method is depicted in an extended version of this paper[1]. We based our study on the popular Eclipse open source integrated development environment [8]. In brief, to answer our research questions we obtained data regarding failures of the Eclipse IDE, we determined the most popular software version associated with the failures, we built this specific software version, we run the provided tests under a code coverage analysis tool, we combined the results with heuristics regarding the naming of test code, we joined the analyzed software failures with the corresponding code coverage analysis results, and we analyzed the results through statistics and a qualitative study. Following published recommendations [10], the code and data associated with this endeavor (AERI JSON data, code coverage analysis, stack traces analyzed, analysis scripts, and combined results) are openly available online.[2].

3 RESULTS

Here we answer our two research questions by means of statistical (RQ1) and qualitative (RQ2) analysis.

3.1 Statistical Analysis

To answer RQ1 regarding the association between unit tests and crashes we classified the 9 523 methods as unit tested and crashed according to the criteria we specified in Section 2.7 of the extended version of this paper[1]. This resulted in their categorization depicted in Figure 3 and and summarized in Table 1.

Our question is whether testing a piece of code is associated with a lower chance of it crashing. Applying Fisher's exact test for count data, results in a p-value of 0.278 and an odds ratio based on the conditional maximum likelihood estimate of 0.915 in a 95% confidence interval of 0–1.146. Consequently, we cannot reject the null hypothesis, and conclude that **our data set does not provide statistical evidence supporting the hypothesis that the presence of unit tests is associated with fewer crash incidents**.

3.2 Qualitative Analysis

To answer RQ2 on why do unit-tested methods still fail, we analyzed the 69 methods that are strictly unit-tested and crashed. This may sound like a small number, but those methods appeared in 10 617 stack traces.

By examining the stack traces and their relevant methods in the Eclipse source code, we classified crashes of unit tested methods into three categories.

1. The method contains a developer-introduced fault. These faults stem from programmer errors, such as algorithmic, logic, ordering, dependency, or consistency errors [14]. They mainly involve code parts that are missing error-handling mechanisms for code that can potentially throw exceptions, thus causing the application to crash with an uncaught exception error.

[1] 10.5281/zenodo.5572888
[2] 10.5281/zenodo.3610822

2. The method intentionally raises an exception. There are methods that intentionally lead to crashes due to internal errors, wrong configuration settings, or unanticipated user behavior, rather than faults introduced through a developer oversight. Developers understood that these failures could potentially happen under unforeseen circumstances or in ways that could not be appropriately handled. As a backstop measure they intentionally throw exceptions with appropriate messages in order to log the failure and collect data that might help them to correct it in the future.

There are methods in the topmost stack position that simply report a failure associated with a fault in another method. These methods are the non-faulty (debugging) methods we described in category 4 of Section 2.6 of the extended version of this paper[1].

Having analyzed the crashes, we worked on understanding why those crashes occurred while there was (apparently) unit tested code. Unit testing would not help alleviate cases 2 and 3, and therefore we did not investigate further. On the other hand, methods belonging to the first case are much more interesting, so we dug deeper to understand the types of faults, failures, and their relationship to unit testing, and categorized them into the following areas.

1.a Method is not called by the class's tests. The method's test class does not call the specific method in any of the tests. The method may have been incidentally called by tests of other classes.

1.b Method is not tested by the class's tests. The method's test class calls this method to setup or validate other tests, but does not explicitly test the given method.

1.c Specific case is not tested. The method has a unit test, but some specific cases are not tested. Ideally, all cases should be tested to ensure that the discrete unit of functionality performs as specified under all circumstances.

4 DISCUSSION

In isolation and at first glance, the results we obtained are startling. It seems that unit tested code is not significantly less likely to be involved in crashes. However, one should keep in mind that absence of evidence is not evidence of absence. We have definitely *not* shown that unit tests *fail* to reduce crashes.

Regarding the result, we should remember that our data come from a production-quality widely used version of Eclipse. It is possible and quite likely that the numerous faults resulting in failures were found through the unit tests we tallied in earlier development, alpha-testing, beta-testing, and production releases. As a result, the tests served their purpose by the time the particular version got released, eliminating faults whose failures do not appear in our data set.

Building on this, we must appreciate that not all methods are unit tested and not all methods are unit tested with the same thoroughness. Figure 1 shows that fewer than half of the methods and lines are unit tested. Furthermore, Figure 2 shows that code coverage within a method's body also varies a lot. This may mean that developers selectively apply unit testing mostly in areas of the code where they believe it is required.

Consequently, an explanation for our results can be that unit tests are preferentially added in complex and fault-prone code in order to weed out implementation bugs. Due to its complexity, such code is likely to contain further undetected faults, which are are in turn likely to be involved in field failures manifesting themselves as reported crashes.

One may still wonder how can unit-tested methods with a 100% code coverage be involved in crashes. Apart from the reasons we identified in Section 3.2, one must appreciate that test coverage is a complex and elusive concept. Test coverage metrics involve statements, decision-to-decision paths (predicate outcomes), predicate-to-predicate outcomes, loops, dependent path pairs, multiple conditions, loop repetitions, and execution paths [12, pp. 142–145], [3]. In contrast, JaCoCo analyses coverage at the level of instructions, lines, and branches. While this functionality is impressive by industry standards, predicate outcome coverage can catch only about 85% of revealed faults [12, p. 143]. It is therefore not surprising that failures still occur in unit tested code.

An important factor associated with our results is that failures manifested themselves exclusively through exceptions. Given that we examined failure incidents through Java stack traces, the fault reporting mechanism is unhandled Java exceptions. By the definition of an unhandled exception stack trace, all methods appearing in our data set passed an exception through them without handling it internally. This is important, for two reasons. First, unit tests rarely examine a method's exception processing; they typically do so only when the method under test is explicitly raising or handling exceptions. Second, most test coverage analysis tools fail to report coverage of exception handling, which offers an additional, inconspicuous, branching path.

It would be imprudent to use our findings as an excuse to avoid unit testing. Instead, practitioners should note that unit testing on its own is not enough to guarantee a high level of software reliability. In addition, tool builders can improve test coverage analysis systems to examine and report exception handling. Finally, researchers can further build on our results to recommend efficient testing methods that can catch the faults that appeared in unit tested code and test coverage analysis processes to pinpoint corresponding risks.

5 THREATS TO VALIDITY

Regarding external validity, the generalizability of our findings is threatened by our choice of the analyzed project. Although Eclipse is a very large and sophisticated project, serving many different application areas, we cannot claim that our choice represents adequately all software development. For example, our findings may not be applicable to small software projects, projects in other application domains, software written in other programming languages, or multi-language projects. Finally, we cannot exclude the possibility that the selection of a specific Eclipse product and release may have biased our results. If anything, we believe additional research should look at failures associated with less mature releases.

Regarding internal validity we see four potential problems. First, the code coverage metrics we employed have room for improvement, by incorporating e.g. branch coverage or mutation testing data. Second, employing JaCoCo on an old release which may have some deprecated code and archived repos, caused some unit test failures, resulting in a lower code coverage. Third, we excluded from

the JaCoCo report non-Java code that is processor architecture specific. Fourth, noise in some meaningless stack frames appearing in our stack trace dataset may have biased the results.

6 RELATED WORK

Among past studies researching the relationship between unit test coverage and software defects, the most related to our work are the ones that examine actual software faults. Surprisingly, these studies do not reach a widespread agreement when it comes to the relationship between the two. More specifically existing findings diverge regarding the hypothesis that a high test coverage leads to fewer defects. Mockus et al. [18], who studied two different industrial software products, agreed with the hypothesis and concluded that code coverage has a negative correlation with the number of defects. On the other hand, Gren and Antinyan's work [9] suggests that unit testing coverage is not related to fewer defects and there is no strong relationship between unit testing and code quality. A more recent study by the same primary author [2], investigated an industrial software product, and also found a negligible decrease in defects when coverage increases, concluding that test unit coverage is not a useful metric for test effectiveness.

Furthermore, in a study of seven Java open source projects, Petric et al. found that the majority of methods with defects had not been covered by unit tests [21], deducing that the absence of unit tests is risky and can lead to failures. On the other hand, Kochhar et al. in another study of one hundred Java projects, did not find a significant correlation between code coverage and defects [15].

The above mentioned studies cover only fixed faults. In our research, we work with stack traces, which enable us to analyze field-reported failures associated with crashes. The associated faults include those that have not been fixed, but exclude other faults that are not associated with crashes, such as divergence from the expected functionality or program freezes. Furthermore, through the crash reports we were unable to know the faulty method associated with the crash. However, by placing our matched crash methods in three groups according to their respective position in the stack trace (in the very first stack frame, within the top-6 and the top-10 stack frames) we could obtain useful bounds backed by empirical evidence [23] regarding the coverage of methods that were likely to be defective.

Considerable research associating testing with defects has been performed on the relationship between test-driven development and software defects. Test-driven development (TDD) is centered around rapid iterations of writing test cases for specifications and then the corresponding code [4]. As a practice it obviously entails more than implementing unit tests, but absence of evidence of TDD benefits should also translate to corresponding absence of benefits through simple unit testing, though the benefits of TDD will not necessarily translate into benefits of unit testing. In a review of the industry's and academia's empirical studies, Mäkinen and Münch [17] found that TDD has positive effects in the reduction of defects a result also mirrored in an earlier meta-analysis [22] and a contemporary viewpoint [19]. In industry, an IBM case study found that test-driven development led to 40% fewer defects [25]. In academia, classroom experiment results showed that students produce code with 45% fewer defects using TDD [7]. On the other hand, experimental results by Wikerson and Mercer failed to show significant positive effects [24].

The study by Jia and Harman [11] shows clear evidence that mutation testing has gained a lot popularity during the past years. The majority of researchers concluded that high mutation score improves fault detection [20]. Furthermore, mutation testing can reveal additional defect cases beyond real faults [1]. However, mutants can only be considered substitute of real faults under specific circumstances [13].

Apart from Schroter and his colleagues [23], a number of researchers have studied the Eclipse IDE and most of them have focused on predicting defects. Most notably, Zimmermann and his colleagues provided a dataset mapping defects from the Eclipse database to specific source code locations annotated with common complexity metrics [27], while Zhang [26] based on Eclipse data, yet again, suggested lines of code as a simple but good predictor of defects.

7 CONCLUSIONS

Software testing contributes to code quality assurance and helps developers detect and correct program defects and prevent failures. Being an important and expensive software process activity it has to be efficient. In our empirical study on the Eclipse project we used the JaCoCo tool and a class source code matching procedure to measure the test coverage, and we analyzed field failure stack traces to assess the effectiveness of testing. Our results indicate that unit testing on its own may not be a sufficient method for preventing program failures. Many methods that were covered by unit tests were involved in crashes, which may mean that the corresponding unit tests were not sufficient for uncovering the corresponding faults. However, it is worth keeping in mind that failures manifested themselves through exceptions whose branch coverage JaCoCo is not reporting. Research building on ours can profitably study the faults that led to the failures we examined in order to propose how unit testing can be improved to uncover them, and how test coverage analysis can be extended to suggest these tests.

ACKNOWLEDGMENTS

We thank Philippe Krief and Boris Baldassari for their invaluable help regarding the Eclipse incident data set. Panos Louridas provided insightful comments on an earlier version of this manuscript. Dimitris Karlis expertly advised us on the employed statistical methods. This work has been partially funded by: the CROSSMINER project, which has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 732223; the FASTEN project, which has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 82532; the GSRT 2018 Research Support grant 11312201; and the Athens University of Economics and Business Research Centre Original Scientific Publications 2019 grant EP-3074-01.

REFERENCES

[1] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th international conference on Software engineering*. ACM, 402–411.

[2] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. 2018. Mythical Unit Test Coverage. *IEEE Software* 35, 3 (May 2018), 73–79. https://doi.org/10.1109/MS.2017.3281318

[3] Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge Profiling Versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 134–148. https://doi.org/10.1145/268946.268958

[4] Kent Beck. 2003. *Test-Driven Development: By Example*. Addison-Wesley, Boston.

[5] Kent Beck and Erich Gamma. 1998. Test Infected: Programmers Love Writing Tests. *Java Report* 3, 7 (July 1998), 37–50.

[6] A. Causevic, D. Sundmark, and S. Punnekkat. 2011. Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. In *Fourth IEEE International Conference on Software Testing, Verification and Validation*. 337–346. https://doi.org/10.1109/ICST.2011.19

[7] Stephen H Edwards. 2003. Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, Vol. 3. Citeseer.

[8] David Geer. 2005. Eclipse becomes the dominant Java IDE. *Computer* 38, 7 (2005), 16–18.

[9] Lucas Gren and Vard Antinyan. 2017. On the Relation Between Unit Testing and Code Quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 52–56.

[10] D.C. Ince, L. Hatton, and J. Graham-Cumming. 2012. The Case for Open Program Code. *Nature* 482 (February 2012), 485–488. https://doi.org/10.1038/nature10836

[11] Yue Jia and Mark Harman. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2010), 649–678.

[12] Paul C. Jorgensen. 2002. *Software Testing: A Craftsman's Approach*. CRC Press, Boca Raton, FL.

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.

[14] Andrew J Ko and Brad A Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1-2 (2005), 41–84.

[15] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. 2017. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* 66, 4 (2017), 1213–1228.

[16] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 560–564.

[17] Simo Mäkinen and Jürgen Münch. 2014. Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies. In *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann (Eds.). Springer International Publishing, Cham, 155–169.

[18] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. 2009. Test Coverage and Post-Verification Defects: A Multiple Case Study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. IEEE, 291–301.

[19] Hussan Munir, Misagh Moayyed, and Kai Petersen. 2014. Considering Rigor and Relevance when Evaluating Test Driven Development: A systematic review. *Information and Software Technology* 56, 4 (2014), 375–394.

[20] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 537–548.

[21] Jean Petrić, Tracy Hall, and David Bowes. 2018. How Effectively Is Defective Code Actually Tested?: An Analysis of JUnit Tests in Seven Open Source Systems. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 42–51.

[22] Yahya Rafique and Vojislav B Mišić. 2012. The Effects of test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Transactions on Software Engineering* 39, 6 (2012), 835–856.

[23] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do Stack Traces Help Developers Fix Bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.

[24] Jerod W Wilkerson, Jay F Nunamaker, and Rick Mercer. 2011. Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development. *IEEE Transactions on Software Engineering* 38, 3 (2011), 547–560.

[25] Laurie Williams, E Michael Maximilien, and Mladen Vouk. 2003. Test-Driven Development as a Defect-Reduction Practice. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 34–45.

[26] Hongyu Zhang. 2009. An Investigation of the Relationships Between Lines of Code and Defects. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 274–283.

[27] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.