On the Differences between Unit and Integration Testing in the TravisTorrent Dataset

Gerardo Orellana
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

Gulsher Laghari
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

Alessandro Murgia
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

Serge Demeyer
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

Abstract—**Already from the early days of testing, practitioners distinguish between unit tests and integration tests as a strategy to locate defects. Unfortunately, the mining software engineering community rarely distinguishes between these two strategies, mainly because it is not straightforward to separate them in the code repositories under study. In this paper we exploited the TravisTorrent dataset provided for the MSR 2017 mining challenge; separated unit tests from integration tests; and correlated these against the workflow as recorded in the corresponding issue reports. Further analysis confirmed that it is worthwhile to treat unit tests and integration tests differently: we discovered that unit tests cause more breaking builds, that fixing the defects exposed by unit tests takes longer and implies more coordination between team members.**

Keywords—Software testing, Integration testing, Unit testing.

I. Introduction

Software testing is the activity of executing a program with the intent of finding a defect. A software test brings the component under test in a given state, then administers a sequence of stimuli and subsequently verifies whether the resulting state corresponds with the expected result. Once a software test exposes a fault, a software engineer still has to locate the root cause of the fault—the *defect*. To minimise the search space, testing handbooks distinguish (among others) between *unit tests* and *integration tests* [1]. A unit test isolates the component under test (typically a class or a method) from the rest of the system so that the tester can be confident that the defect is located within the unit. An integration test, on the other hand, exercises the interfaces between units; when an integration test exposes a fault the defect should be in the code that implements the protocol between the units.

Given that unit and integration tests represent different strategies to pinpoint the location of a defect, one would expect that software engineers take different actions when confronted with a failing test. In particular, we assume that unit tests expose more defects since they come earlier in the testing cycle. Also, resolving a failing integration test should take longer, since the search space for locating the defect is larger. Similarly, resolving a failing integration test should require more coordination between team members, because more software engineers have been involved in the implementation.

However, with the advent of continuous integration, the software testing landscape changed drastically [2]. Given a continuous integration setting, the sharp distinction between unit tests and integration tests is disappearing; they are all tests to support the team in writing code without worrying about making unintended changes to the system [3].

The TravisTorrent dataset used for the MSR2017 mining challenge posed an ideal testbed for verifying whether there is indeed a difference between unit- and integration testing in continuous integration setting. TravisTorrent is a freely available dataset synthesised from the Travis Continuous Integration Server and GitHub. Via this dataset, it is possible to see which builds contained a defect, which tests have been able to expose the defect, and inspect the actions the software engineers took for repair. We selected 423 projects within the complete dataset where we could distinguish between unit tests and integration tests and where we could rely on information available in JIRA or GitHub to calculate the time to fix. For these projects we recovered those tests which exposed a defect, classified those tests in either unit or integration tests, and mined the issue tracker (JIRA or GitHub) and the version control system (GitHub) to inspect the resolution actions.

Our work makes the following contributions:

1) We search 423 java projects in TravisTorrent dataset to find the defects and categorise the failing tests into unit tests and integration tests.
2) We analyse the defects exposed by unit tests versus integration tests and discover that unit tests cause more builds to fail.
3) We compare the time to fix the defects exposed by unit tests against integration tests and discover that fixing defects exposed by unit tests takes longer
4) We contrast the different actions software engineers take when resolving failing unit tests versus integration tests and discover that unit tests involves more coordination between team members.

II. Research Questions

The goal of this work is to gain insight into two supposedly different testing strategies —unit testing versus integration testing— in a continuous integration setting. We analyse the repositories of projects in TravisTorrent and their corresponding issue tracking systems (JIRA or GitHub) in a quantitative way in order to answer the following research questions:

RQ1 – *Do unit tests expose more defects than integration tests?*

Motivation: In the normal testing life-cycle, unit testing precedes integration testing because one should first rule out the defects *within* a unit before one proceeds to find defects *between* units [1]. A good testing strategy aims to find defects as early as possible, hence the aspiration is that unit tests find more defects than integration tests. However, in a continuous integration setting both unit tests and integration tests are coded in the same testing harness —usually a variant of xUnit— and executed simultaneously [3]. Thus, unit tests and integration tests are treated similarly in a continuous integration setting, hence it remains to be seen whether there is indeed a difference.

RQ2 – *Does the resolution of a defect exposed by an integration test take longer than that of a unit test?*
Motivation: An integration test exercises the interfaces between units, thus the defect is supposedly located in the code that implements the protocol between the units. Unfortunately, one cannot entirely rule out the code within the units, thus the location of the defect is potentially in all components involved in the integration test. Since, the search space for locating the defect exposed by an integration test is larger, we may expect that it takes longer to resolve. Yet, here as well, the practice of continuous integration is slightly different: both unit tests and integration tests are meant to gain confidence and the tests are meant to pass with every build. When a test fails, the search space is restricted to the recent changes, thus it is small in either case.

RQ3 – *Does the resolution of a defect exposed by an integration test involve more coordination between the software engineers than that of unit test?*
Motivation: Once a test exposes a defect, it must be assigned to a software engineer for resolution. For a failing unit test it is rather straightforward: it is one of the software engineers who recently worked on the unit under test. However, for a failing integration test it is not as straightforward since it may be any of the software engineers implementing either the unit under test or one of the dependent components. Thus, one would expect that resolving a failing integration test entails more coordination among the software engineers involved. However, in a continuous integration setting it is the issue tracking system which drives the work within a team and it remains to be seen whether unit test or integration test affects the workflow.

III. Complementing the Dataset

For this investigation we started from the TravisTorrent dataset [4], which we complemented with data gleaned from GitHub as well as JIRA. TravisTorrent contains the history of build information of 1.359 open source projects written in ruby and java; To complement the TravisTorrent dataset, we took a series of actions detailed in the subsequent paragraphs.

Select Projects. For this investigation, we restricted ourselves to 423 of the java projects where we could distinguish between unit tests and integration tests and where we could rely on information available in JIRA or GitHub to calculate the time to fix.

Identify Defect Fixing Commits. To identify the *defect fixing commits*, we distinguish between projects using GitHub or JIRA as their issue tracking system, which determines the search patterns to use in the project commit messages. We download the repositories of 423 java projects from GitHub. Then, we use JGIT [5] to walk though the commit history of a project similar to Vahabzadeh et. al. [6]. We label a commit as *defect fixing commit*, if it contains the issue ID coupled with phrases like *fixes* or *closes*. Therefore, we search in the commit messages for text patterns combining the issue ID with words like *fixes*, *fixed*, *fix*, *closes*, *closed*, or *close*. We repeat this process for each project to arrive at the database of projects with their fixed defects.

Identify Failing Tests. Once we obtained a dataset of *defect fixing commits*, we collect the names of those tests that expose the defect by looking for the corresponding failing build in the TravisTorrent dataset. We combine two approaches: on the one hand, we analyse the logs of the builds provided by TravisTorrent to see what precisely breaks the build, on the other hand we simply use the failing test names (column *tr_failed_tests*) available in the TravisTorrent dataset.

Distinguish between Unit Tests and Integration Tests. We combined two heuristics to distinguish unit tests from integration tests, one from information gleaned from the test environment itself, the other one based on naming conventions within the test code.

We looked into the use of the SureFire and FailSafe plugins with Maven build system [7], [8]. SureFire corresponds with a unit test, while FailSafe corresponds with an integration test. Of course this only works for those projects that relied upon Maven to build, hence this reduced the projects under consideration.

To complement the former, we exploited naming conventions adopted from the ones specified by Appel [9]. These state that the class name of a unit test has the word “Test” as a suffix (sometimes as a prefix) of the name of the class under test. For example the unit test that tests a class named “SomeClass” would be “SomeClassTest”. Similarly, the name of the integration test contains the word “IntegrationTest”. However, via manual inspection, we learned that software engineers also use acronyms or abbreviations of ”IntegrationTest”, such as “IT”, which we also incorporated.

Using this process, we identified 9.118 tests of which 8.382 are unit tests (91.93%) and 736 (8.07%) are integration tests. As can be expected the number of unit tests is significantly higher than the number of integration tests.

After categorising the tests in unit tests and integration tests, we classify the failing builds into three different categories.

1) *unit tests* when the set of failing tests contains only unit tests,
2) *integration tests* when the set of failing tests contains only integration tests, and

3) *both* when the set of failing tests contains both types of tests.

Calculating Time to Fix. We calculate the *time to fix* of a given defect as the duration between the time when an issue is assigned to a software engineer until it is closed (*Closed Time - Assignation Time*). This is a reasonable proxy for the actual time spent in resolving an issue and is done by other researchers as well [10], [11], [12]. Thus, for each defect in our dataset, we collect the following information from the projects' corresponding issue tracking system.

Creation Time — The datetime when the issue was created.

Assignation Time — The datetime when the issue was assigned to a software engineer: it is useful to determine the resolution time of the issue.

Closed Time — The datetime when the commit closed the issue giving it the state of closed or resolved.

Assessing Team Coordination. Finally, we assess the amount of team coordination required for fixing a defect from the following fields in the issue tracker.

Number of comments — The number of comments by software engineers on the discussion about resolving the issue.

Number of Unique Software Engineers — The number of unique software engineers involved in any type of event related to resolving a defect.

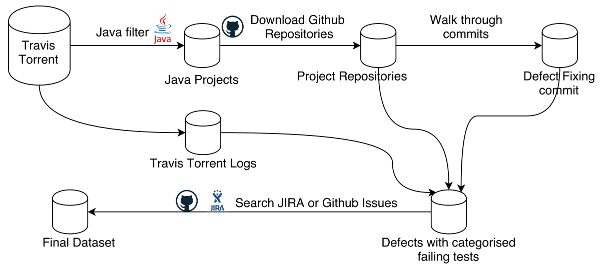


Fig. 1. The process to complement the TravisTorrent dataset

Figure 1 illustrates the process we followed to obtain the final dataset. We found 79.078 *defect fixing commits* from project repositories. Out of these 79.078 *defect fixing commits*, there were only *16.203* where we could link the *defect fixing commit* to a *failing build* in TravisTorrent. For 14.959 of these *defect fixing commits*, there were no build logs available. Thus there we could not identify the tests which exposed the defect, hence were removed from the dataset. Eventually, this resulted in 1.244 defects which we could link to failing tests. 5 of these had the same *Assignation Time* and *Closed Time* —a practice which occurs when software engineers first fix a defect and only afterwards file an issue report— hence we excluded these 5 defects from our final dataset.

At the end of complementing the TravisTorrent dataset, we arrived at 1.239 defects linked to one or more failing tests and where we have the *time to fix* available. Table I shows details of our final dataset.

TABLE I. DATASET DETAILS

#Projects	$\#UT_{defects}$[†]	$\#IT_{defects}$[‡]	$\#Both_{defects}$[*]	Total Defects
19	683	454	102	1.239

[†] # defects exposed by unit tests [‡] # defects exposed by integration tests
[*] # defects exposed by both unit tests and integration tests

TABLE II. TESTS DISTRIBUTION

$\# UT_{number}$	$\# IT_{number}$	Tests Total
8.382	736	9.118

IV. FINDINGS

RQ1. Table I lists the number of defects exposed by unit tests, versus the amount exposed by integration tests versus the ones exposed by both an integration and a unit test.

The dataset contains more defects exposed by unit tests than integration tests. Out of 1.239 defects, 683 (55%) are exposed by unit tests, while 454 (37%) are exposed by integration tests and only 102 (8%) are exposed by both unit tests and integration tests together. This indicates that more defects are exposed by unit tests than integration tests. This may be partly explained by the distribution of the tests shown in Table II; since there are significantly more unit tests than integration tests.

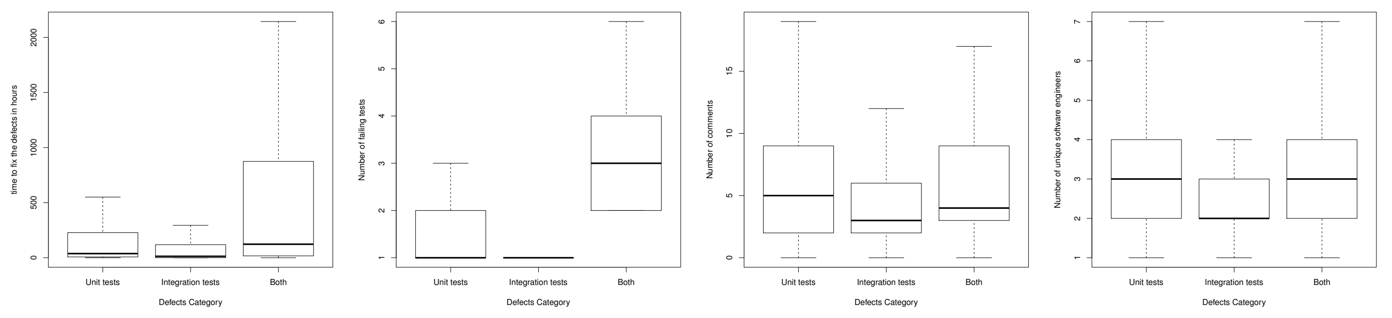
Unit tests expose more defects than integration tests. The phenomenon is most likely due to effect size as there are significantly more unit tests than integration tests.

RQ2. Figure 2a shows the distribution of *time to fix* the defects in each category namely defects exposed by unit tests, integration tests, and both unit tests and integration tests together.

By analysing the distribution of the *time to fix* in each category, we find that defects exposed by unit tests have relatively longer *time to fix* than those exposed by integration tests. However, the time is comparatively longer when the defects are exposed by both unit tests and integration tests together. Given the non-normal distribution of the *time to fix*, we conducted a Kruskal-Wallis Test resulting in a p-value of 0.045. confirming that our results are statistically significant.

To reason about and explain further the variation in *time to fix* the defects exposed by unit tests, integration tests, and both unit tests and integration tests together, we also analyse the number of tests failing for each defect. Figure 2b shows the distribution of the number of failing tests in defects for each category. We observe that when a defect is exposed by integration test, it is always only one integration test failing. However, when the defect is exposed by unit tests or both unit tests and integration tests together, there are more failing tests. This might also make it difficult to locate the defect, hence increased *time to fix*. The Kruskal-Wallis Test resulted in a p-value of 1.369e-05, confirming that the *time to fix* is greatly affected by the number of failing tests.

It takes more time to fix the defects exposed by unit tests than those exposed by integration tests. However, when both unit tests and integration tests fail together, the time to fix the defects is even higher.



(a) Distribution of *time to fix* the defects (b) Distribution of failing tests (c) Distribution of comments (d) Distribution of software engineers

Fig. 2. The boxplots showing distribution of various defect related factors without outliers

RQ3. Figure 2c shows the distribution of the number of comments involved in resolving the defects for each category. We see that software engineers engage in more discussions when resolving defects exposed by unit tests compared to those exposed by integration tests.

As we observed in Figure 2b; the number of failing tests is larger when defects are exposed by unit tests or both unit tests and integration tests together. We assume that since there are more failing tests, software engineers discuss more to understand the defect. Hence, the amount of comments is higher when resolving the defects exposed by unit tests. This also explains why it takes more time when resolving the defects exposed by unit tests (Cf. **RQ2**). Since resolving the defects exposed by unit tests takes more discussion amongst software engineers, this may cause increase in *time to fix* the defects. The Kruskal-Wallis Test over the number of comments showed a p-value of 0.1183. Thus, we conclude that the variation in comments for each category seems important even though it is not statistically significant.

In addition, we also explore the number of software engineers involved in resolving a defect, shown in Figure 2d. Here we also observe that resolving the defects related to unit tests involves more individuals than those of integration tests. Calculating the statistical significance, we obtained 0.013. We used the Fisher's method for combining the p-values of the number of comments and the number of unique authors involved and got a resulting p-value of 0.0115. This leads to the conclusion that the involvement of the community is strongly differentiated in every category of tests.

There are more software engineers involved in resolving the defects exposed by unit tests than those exposed by integration tests and it requires more coordination.

V. Conclusion

In this paper, we analyse the repositories of projects in TravisTorrent and their corresponding issue tracking systems (JIRA or GitHub) in a quantitative way in order to gain insight into the differences between unit testing and integration testing in a continuous integration setting. Our results show that for the 1.239 defects under investigation, unit tests indeed cause more builds to fail. However, contrary to our expectations, fixing defects exposed by unit tests takes longer and involves more communication between team members. These results suggest that the MSR community should be careful when analysing how software teams handle defects. When mining for actionable information from test code, researchers should, at least, distinguish between unit- and integration tests.

References

[1] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[2] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *Leaders of Tomorrow: Future of Software Engineering, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[3] L. Crispin and J. Gregory, *Agile testing; a practical guide for testers an agile teams*. Addison-Wesley, 2009.

[4] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[5] "Jgit library," https://eclipse.org/jgit/.

[6] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 101–110.

[7] "Maven surefire plugin," http://maven.apache.org/components/surefire/maven-surefire-plugin/, accessed: 2017-02-08.

[8] "Maven failsafe plugin," http://maven.apache.org/surefire/maven-failsafe-plugin/, accessed: 2017-02-08.

[9] F. Appel, *Testing with Junit*, ser. Community experience distilled. Packt Publishing, 2015. [Online]. Available: https://books.google.be/books?id=NzsbjgEACAAJ

[10] S. N. Ahsan, M. T. Afzal, S. Zaman, C. Guetl, and F. Wotawa, "Mining effort data from the oss repository of developers bug fix activity," *Journal of IT in Asia*, vol. 3, pp. 67–80, 2010.

[11] P. Ramarao, K. Muthukumaran, S. Dash, and N. L. B. Murthy, "Impact of bug reporter's reputation on bug-fix times," in *2016 International Conference on Information Systems Engineering (ICISE)*, April 2016, pp. 57–61.

[12] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 1.