Continuous Integration Build Failures in Practice

by

Adriaan Labuschagne

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Automated software testing is a popular method of quality control that aims to detect bugs before software is released to the end user. Unfortunately, writing, maintaining, and executing automated test suites is expensive and not necessarily cost effective.

To gain a better understanding of the return on investment and cost of automated testing, we studied the continuous integration build results of 61 open source projects. We found that 19.4% of build failures are resolved by a change to only test code. These failures do not find bugs in the code under test, but rather bugs in the test suites themselves and represent a maintenance cost. We also found that 12.8% of the build failures we studied are due to non-deterministic tests that can be difficult to debug. To gain a better understanding of what leads to test maintenance we manually inspected both valuable and costly tests and found a variety of reasons for test maintenance.

Our empirical analysis of real projects quantifies the cost of maintaining test suites and shows that reducing maintenance costs could make automated testing much more affordable.

Acknowledgements

I would like to thank Reid Holmes, Laura Inozemtseva, and Cheng Zhang for all of their help as well as Alexandra Roatiş, my parents, and sister for putting up with all my nonsense.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automated regression testing is a common technique used to catch faults during the software evolution process before they reach the end user. Tests can also provide secondary benefits such as API usage examples [15] and increase developer confidence during refactoring. Automated testing techniques, however, have many associated costs. These include writing the tests, maintaining the tests as the system evolves, executing the tests, and inspecting test results. Beller et al. [2] recorded developers testing behaviour in their IDEs and found that they spend a quarter of their time reading and writing tests. These costs can be perceived as prohibitive. One participant in an industrial study [7] was quoted as saying, "Developing that kind of test automation system is almost as huge an effort as building the actual project".

The high cost of automated testing has led to efforts to optimize the process, including test prioritization, test selection and test suite reduction [3, 6, 8]. Test prioritization techniques run tests that are more likely to fail first so that developers can start debugging issues as soon as possible while the code changes are still fresh in their minds (some systems take as long as three days to run [1]). Test selection techniques run only a subset of all tests that are considered the most likely to find faults. Test selection is attractive because even executing all tests on a regular basis can be expensive. A study by Herzig et al. [6] found that their test selection technique could save Microsoft $1.6 million in test execution costs over a 26 month period of Microsoft Windows development. Finally, test suite reduction techniques identify tests that overlap with other tests, and therefore do not increase the fault finding ability of the test suite. Permanently removing these tests reduces both execution and maintenance costs. Others studies [20, 16] have examined the maintenance costs of automated testing by studying test suite evolution through mining version control systems. These studies can, for instance, determine what portion of tests

required maintenance between product versions. What these studies have not done is provide an idea of the value tests provide compared to the maintenance costs of tests. In other words, what is the ratio between faults found in the system under test and faults found in the test suite itself?

This thesis explores the return on investment provided by automated test suites in a continuous integration environment. Specifically, we address the following research questions:

Research Question 1 *What portion of test suite failures represent a return on investment?*

Research Question 2 *What portion of test suite failures are non-deterministic?*

Research Question 3 *How often do tests fail alone?*

Research Question 4 *How does the maintenance cost of test suites, and individual tests compare within, and across projects?*

To answer these research questions we gathered data from Travis CI, a continuous integration service that has become popular with open source software projects. Travis CI executes tests in the cloud when a developer pushes new commits to their online repository (e.g., a repository on GitHub). This frees up developer machines and provides an identical test infrastructure making failures more reproducible. These benefits lead many developers to push untested changes that result in failed Travis CI builds. Travis CI build failures provide a unique opportunity in that the changes that caused the failure as well as the changes that eventually fix the failing tests can be found in the project repository; these changes are ordinarily lost in an offline workflow since committing broken code is frowned upon.

We discuss related work in Chapter 2, present our methodology in Chapter 3 and our results in Chapter 4.

Chapter 2

Related Work

The work presented in this thesis is closely related to the areas of test evolution, test failures, test maintenance, and the cost effectiveness of regression testing. In this section we describe a number of projects representing the state of the art in each of these categories.

2.1 Test Evolution

Production code must evolve continuously to remain useful [10]. This implies that test code must also change as new features require new tests, while updates to existing functionality, or even refactoring [14], will require old tests to be repaired.

Zaidman et al. [20] investigated the co-evolution of test and product code by mining source control systems and code coverage reports. They introduced test co-evolution views: visualizations to help developers answer questions about the evolution of their product and test code. Their visualizations were evaluated on two open source projects, revealing both close synchronous evolution as well as separate, stepwise evolution. On a larger selection of projects, Beller and Zaidman [2] later found that although production and test code tend to change together it is far from a strict rule.

Marsavina et al. [12] studied how developers evolve their production and test code. They found that test code changes made up between 6% and 47% of all code changes for the projects they studied. This indicates that testing can be a major part of development effort but they did not investigate what portion of this effort benefits the project.

Pinto et al. [16] studied test evolution to see how tests are repaired in practice so as to lay the groundwork for future studies on automated test repair. They found that 29.6%

of changes made to test suites were test modifications, 15% were test additions, and 56% were test deletions. The 29.6% was further split into repairs that fix broken tests (22%) and changes that were made to tests that pass even without the changes (78%). They also recorded the type of code change (e.g., parameter deletion) used to repair tests.

Our work differs from the above studies in that (1) we tie test evolution to test suite execution results (2) we study a larger number of open source systems and (3) we study test results and changes at the build level rather than at the level of software versions.

2.2 Test Failures

Gaining an understanding of when test cases are executed as well as how they fail has been studied in a number of different contexts. These studies provide insight into how developers handle test failures.

Some test failures are caused not by an error in the system under test (SUT) but rather by a change in functionality or even a refactoring of the SUT. When this happens the test suite has to be modified to account for the change. This is purely a maintenance cost as the test did not discover a bug in the system under test. Graphical user interface (GUI) tests are particularly sensitive to this type of failure. An example is a test that needs to click on a button in order to verify the outcome. If the button is moved to another menu, any test that refers to it may no longer be able find the button causing them to fail even though the product still functions correctly. Memon and Soffa[13] found that 74% of GUI test cases may become obsolete between two versions of the industrial application they studied. To address this situation they developed a technique to automatically repair GUI tests that become un-runnable because of layout changes. Their technique was able to repair more than 70% of the test cases that became obsolete during their study.

Beller et al. [2] recorded the development behaviour of developers by monitoring their actions with an Eclipse plugin called WatchDog. This allowed them to record the tests developers execute in their IDE and the time they spend reading and writing test and production code. They found that although developers believe they spend half their time on testing they in fact spend only about a quarter of their time on testing (i.e., reading or writing test code). Test driven development was also less common than developers claimed; it could only be detected to a limited degree in 12% of projects that claim to follow it. They also found that many developers do not test at all (in 57% of projects no reading, writing, or execution of tests were detected), those that do execute tests typically execute a small number of tests at a time (87% of test runs contain only one test case), and they

failed to record a successful execution for 30% of failing tests. While Beller et al. [2] focus on testing in the IDE and the effort spent on testing, our work explores the link between the changes that cause and resolve continuous integration build failures.

Anderson et al. [1] developed two test prioritization techniques that utilize past results. The first technique, *most common failures*, suggests that tests that failed the most in the past (e.g., in more than 50% of test runs) are most likely to fail in the future. The second, *failures by association*, attempts to use association rule mining to improve on the first. They find that the techniques have similar performance when predicting future failures (F-measure of 0.55) and that both work better when using a small window of recent executions, rather than on the entire historical dataset.

Luo et al. [11] investigated 201 commits from 51 projects that likely fix flaky tests (i.e., non-deterministic tests). They classified the root causes of flakiness into ten categories. They further found that 77% of the flaky tests they inspected belong to one of the top three categories, namely Async Wait (i.e., tests that do not properly wait for asynchronous calls), Concurrency, and Test Order Dependency. They also investigate methods to help identify flaky tests such as changing or adding `sleep` methods to expose Async Wait tests. Finally, they categorized flaky test fix types into twelve categories.

Vasilescu et al. [19] explored the relation between Travis CI build results (success or failure) and the way the build was started (i.e., direct commit from a developer with write access to the repository or a pull-request). They found that builds started by pull-requests are more likely to fail than those started by direct commits. They also found that although 92% of the projects they studied are configured to use Travis CI, only 42% actually do. We also found that many projects signed up for Travis CI but do not actively use it as explained in Section 3.2.

2.3 Test Maintenance

Changes to product code often necessitate changes to test code. These product code changes could be functionality preserving, such as code refactoring, or changes that alter the functional properties of the system. Test maintenance is a large part of the overall cost of automated testing and needs to be taken into account when new tests are written.

Grechanik et al. [4] created a tool that uses static analysis to detect test script references that are affected by changes in the applications GUI (e.g., a series of checkboxes being replaced by a dropdown). The tool generates warnings that allows developers to fix these errors before running the test scripts, thereby saving valuable time.

Hao et. al. [5] attempt to detect whether a test failure is due to a bug in the code under test or to a fault in the test itself. They do this by training a classifier on seven features related to test failures. These include the maximum depth of the call-graph of a test, a deep call-graph indicates a complicated SUT which they argue makes it likely that the fault is located in the SUT. Another example is whether or not a failing test calls a method that is called by more than half the failing test, if this is the case the method in question is likely to blame for the failure. To train the classifier they needed a large number of faulty tests as well as tests that fail because of a bug in the SUT. To find faulty tests they ran the test suite from an older version of the program on a newer version. A test that fails under these circumstances likely had to be changed at some point in order to run on the new version and this failure can therefore be viewed as a test bug. To find examples of bugs in product code that cause test failures the authors mutated the code under test, any resulting test failure was seen as due to a bug in the code. Their approach classified 96.15% of failures accurately when training and testing on different versions of the same program. When used to classify failures between two different programs the accuracy decreased to 71.15%.

2.4 Cost Effectiveness of Regression Testing

Many studies have focussed on improving the cost effectiveness of regression testing. The most common approaches are regression test selection (RTS), test suite reduction, and test suite prioritization. RTS techniques aim to select the subset of tests that is the most likely to find faults, thereby reducing test suite execution time. Test suite reduction techniques aim to permanently remove tests that may, for instance, be redundant because they overlap with other tests. Test suite prioritization techniques execute tests that are more likely to fail first so that faults may be found earlier.

Herzig et. al. [6] developed a test selection tool called THEO that selects tests to be executed if the probable cost of execution is lower than the probable cost of skipping the test. To calculate these costs THEO needs the past defect detection probability for each test case (i.e., true positive rate P_{TP}) and the past false alarm probability of a test case (i.e., false positive rate P_{FP}). These probabilities can then be used along with data such as machine costs, number of engineers and inspection costs in order to calculate the cost of skipping and executing a test. An evaluation of the system on historical data from Microsoft allowed 35% to 50% of tests to be skipped while only letting 0.2% to 13% of defects escape.

Elbaum et al. [3] used test selection at the pre-submit stage and test prioritization

during the post-submit stage to increase the cost effectiveness of testing. During the pre-submit phase (i.e., before code is pushed to the central repository) tests that failed during a pre-determined failure window are selected to be executed, the basic intuition being that recent failures likely predict future failures. New tests and tests that have been skipped more than a set number of times are also selected. During the post-submit testing phase, one wants to avoid skipping test suites and therefore tests are prioritized, rather than selected, based on the same criteria used for selection during the pre-submit phase. Test prioritization ensures that all tests are executed during the post-submit phase while running tests that are most likely to fail first, thereby shortening the feedback loop. Elbaum's technique is simpler than the technique used by THEO since many inputs such as number of engineers and machine costs are not required. The technique also performed better than THEO by selecting fewer tests (27%) while still letting fewer failing tests escape (0.17%).

Most test suite reduction techniques use coverage to detect redundant tests. This leads to a loss in test suite fault detection ability because fully overlapping coverage does not necessarily mean the tests will always fail under the same circumstances [9]. To address this issue Koochakzadeh and Garousi [8] developed a test reduction tool that brings a human tester into the loop. Their tool identifies potentially redundant tests using coverage analysis, and then lets testers inspect these tests to identify the true positives that can be removed from the suite. Using this technique they were able to remove eleven of the 82 tests in their subject system without affecting the suite's mutation score. In contrast the coverage based approach identified 52 tests as redundant and reduced the mutation score by 31%.

The effect of test suite granularity on the cost effectiveness of regression testing was studied by Rothermel et al. [17] Test suite granularity refers to the number of statements or number of inputs per test. In their study, Rothermel et al. [17] found that by making tests four times larger they could reduce test suite execution time by 73% because of savings in test setup cost. This is an even larger saving than the regression test selection technique they applied to the same test suite, which reduced execution time by only 60%. A drawback of larger tests is that it likely makes fault localization more difficult.

Chapter 3

Methodology

The goal of our study is to improve developers' understanding of the costs of automated testing, particularly when used in a continuous integration environment. To do this, we needed a source of test results (Section 3.1) from an appropriate set of projects (Section 3.2) and we needed to link test results to the changes that influenced their outcome (Section 3.3). In order to determine the costs and benefits of test failures in practice we also needed a method to distinguish between test failures that indicate a fault in the production code and those that indicate a need for test suite maintenance (Section 3.4). Lastly, we describe how we measured the prevalence of flaky test suites and their effect on our model of test suite maintenance (Section 3.5) and how we parsed Travis CI logs to extract the names of failing tests (Section 3.6).

3.1 Collecting Test Results

The primary obstacle to studying test failures in practice is that most developers run their tests locally before commiting their code. This means that the test results, as well as the changes associated with those results, are lost. Some of these lost test results can be recovered from continuous integration services, such as Travis CI, that execute unit tests in the cloud. Travis CI builds a project and executes its tests every time a developer pushes a change or opens a GitHub pull request. A Travis 'build' consists of a set of one or more commits; for example, a pull request can be composed of three commits that are executed as a single build. Travis CI offers a number of advantages over a traditional non-cloud based workflow: first, it ensures all developers on a team have identical test infrastructure making failures more consistently reproducible; second, TravisCI can execute tests on a variety of

platforms (e.g., operating systems) and library versions (e.g., Java6, Java7, and Java8) concurrently, making it easier to detect faults that occur only on certain configurations; and third, by having the test status visible to all project members (and the public), a team can create a greater awareness of code quality.

The ease of running tests in the cloud has led many developers to commit their changes and push them to the remote repository, resulting in Travis executing a build, without running the test suite locally. This means that failure-inducing changes that would have been fixed (in a non-cloud workflow) before being committed can now be found in the repository and linked to the failing build.

Travis stores a state for each build that can have the following values:

- *Pass*: The build was successfully compiled and all tests passed.
- *Error*: The build failed before test execution began, i.e., there was a compilation or configuration error.
- *Fail*: The build was successfully compiled, but one or more test assertions failed or an unexpected runtime exception was encountered.
- *Cancel*: A developer manually terminated the build while it was running.
- *Started*: The build was started but has not finished yet.

Travis also stores, and makes available through its API, the branch and commit identifier (SHA) of the head commit associated with the build, allowing us to link build results to code change sets.

3.2 Subject Program Selection

We selected subject programs for our study by querying the GitHub Archive[1] for Java projects that received more than 1,000 push events between 2012 and 2014. We chose this time frame because Travis support for Java, which began in February 2012. The query returned 685 projects; of these, 421 projects had Travis accounts. We were able to successfully clone 402 of these projects from GitHub. Since the focus of our analysis was on regression testing, we eliminated early-stage projects. To do this, we removed subjects that

[1] https://www.githubarchive.org/

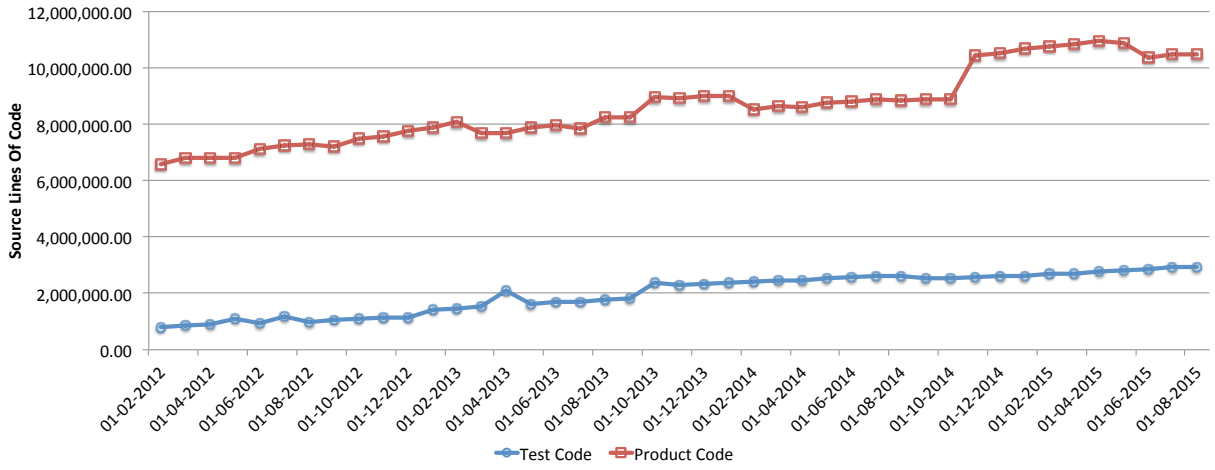


Figure 3.1: Growth of production source code and test source code over the study period.

were less than two years old, resulting in 362 systems. Of these, only 101 projects actively used Travis to execute their test suites; the other 261 projects had signed up for the service but did not use it. Unfortunately, several of these remaining projects did not configure Travis correctly or did not examine the Travis output, resulting in long stretches of broken builds; others almost never experienced a failure. To account for this, we removed the 20% of projects that had the most errors and failures and the 20% that had the fewest errors and failures, resulting in a final set of 61 projects.

3.2.1 Test Suite Size

It is important to get a sense of the amount of effort the 61 projects we studied invested in testing. To do this, we counted [2] the number of lines of test and production code in each project for every month between February 2012 and August 2015. In August 2015 the projects contained over 2.9 million lines of test code and 10.4 million lines of production code. The results for each month, aggregated across projects, are shown in Figure 3.1, where it can be seen that both test and production code exhibit relatively steady growth throughout the study period. Although both test and production code grew over the study period, test code grew faster increasing its portion of the total code from 10.8% to 21.9%. Both the large portion of test code and the continued growth in the portion of test code indicates that testing is important to the authors of the 61 projects.

[2]Counted using non-comment source lines (NCSL) using `cloc` https://github.com/AlDanial/cloc.

3.3 Identifying Changes Associated with Builds

In order to identify the changes that caused and resolved build failures three commits were required: the head commit of the build before the failure, at the failure, and after the failure. Travis CI stores a commit SHA (the SHA of the head of the branch being built) for each build, and all of the projects we studied were downloaded from Github making these commits available for analysis. Unfortunately, the commit associated with a build was not always present in the repository. There are two reasons for this:

1. The build was associated with a pull request. Travis creates a new build every time a pull request is created or updated. The commits associated with these builds, however, are never present in the repository. However, once the pull request is accepted and merged, another build is run on the merge commit that can be found in the repository.

2. History rewrites. If a developer rewrites history, the builds associated with the rewritten commits will no longer be traceable to a commit in the repository.

The 61 projects we studied ran 106,738 builds on Travis during our analysis period. Of these, 70,448 (66%) builds had an associated commit in the version control repository of their projects. This reduced dataset was used for analyses that required access to the source code associated with the build (e.g., Section 3.4) while the full data set (106,738) was used for other analyses.

3.4 Analyzing Build States And Transitions

We consider there to be three ways for a test suite to provide a return on investment:

1. A test fails and indicates a regression, i.e., a fault in the production code that needs to be repaired. This test failure provides value by preventing the bug from reaching end users.

2. A test passes, increasing the confidence of the developers. This may, for example, result in faster refactoring because developers can feel confident that they have not introduced an error.

3. Tests can serve as documentation, providing examples of API usage [18].

Unfortunately, not all test failures identify a fault in production code. It could also be the test itself that is at fault. These failures represent a maintenance cost rather than a return on investment. In our work we aim not only to determine the frequency of failures in practice but also to separate test failures that provide a return on investment from those that indicate a need for maintenance. To do this, we study the code changes that cause a build state to transition from passed to failed and back to passed again.

Such a `Pass` → `Fail` → `Pass` tuple can be seen in Figure 3.2. Each box in the diagram represents a commit and each box is labeled with the type of code changed by the commit. A commit that changed only production code is labelled with a 'C'. A 'T' is used to indicate a commit that changed only test code. Finally, a commit labeled with a 'CT' changed both production and test code. We assumed a file to be a test file if the filename or path contained the word 'test' and the file had a code extension. To determine what counts as a code extension, as opposed to something like `.txt`, we went through a list of all the extensions found in our subjects and manually classified them as code or non-code. The full list can be found in Appendix B.1.

Travis executes a new build every time code is pushed to Github. This code is often made up of several commits that are built together. As an example, in Figure 3.2 the first build passed and was made up of two commits; one that changed only the production code, and another that changed both production and test code. Next, the developer pushed three commits containing changes to source code only, and the build this push triggered failed. To get the set of changes introduced between this failing build and the successful build before it, we take the diff between the last commit in the failing build and the last commit

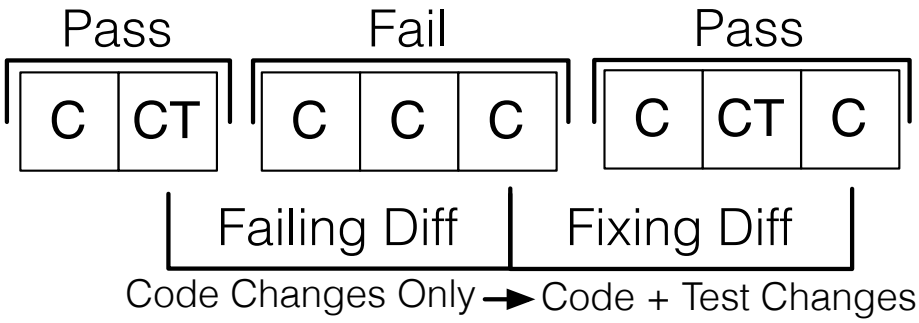


Figure 3.2: The figure shows a failure caused by a code change that was resolved by fixing both code and test files. Each box represents a commit; builds are often not run on every commit but instead on blocks of commits. A C label on a commit means the code under test was changed; a T label means the test code was changed.

Table 3.1: Categorization of how builds transition between Pass and Fail states. The left-hand-side of → denotes the resources that changed to cause the test failure. The right-hand-side of → denotes the resources that were changed to resolve the test failure.

Fix Type	**Resources Changed**
Code fixes	Code → Code
	Test → Code
	Code+Test → Code
Test fixes	Code → Test
	Test → Test
	Code+Test → Test
Mixed fixes	Code → Code+Test
	Test → Code+Test
	Code+Test → Code+Test

in the passing build; in this example, only production code changed. The developer now pushed to the repository a third time, triggering a third build consisting of three commits. This build passed and again we take the diff between the last commit in this build, and the last commit in the build before it; this time to determine what type of changes fixed the build. In this case, both production and test code changed, making it unclear what type of change resolved the issue.

We chose to limit our analysis to `Pass` → `Fail` → `Pass` transitions that include exactly one failed build between the passed builds for two reasons. First, it is the most common type of tuple (1,381 tuples compared to 2,170 tuples if allowing any number of failures). Second, allowing any number of failures would include cases where the build fails twenty times in a row. In such a case it is likely that the developers were not trying to fix the issue every time they pushed code and therefore seeing all 20 builds as part of the fix would be inaccurate. At the same time, assuming that it was only the last of the 20 builds that tried to fix the issue is also likely to be inaccurate.

Table 3.1 describes all the possible transitions between three builds that constitute `Pass` → `Fail` → `Pass` tuples. The builds have been grouped by the kind of fix that resolves the build failure. In our analysis we considered all failures with code-only changes to be a return on investment and those fixed by test-only changes to be a maintenance cost.

3.5 Examining Test Inconsistencies

While manually inspecting a number of `Pass` → `Fail` → `Pass` tuples we noted that the changes 'breaking' and 'fixing' builds often seem unrelated to the failing tests. A good example of this are commits 1494447 and 92679e8 from Apache PDF Box [3] that respectively break and fix the build despite only reformatting author tags. The non-determinism was eventually explained in a bug report [4] and fixed in commit 6dc4a35.

To determine a lower bound for test suite flakiness we used the Travis API to rerun the failed build for each of the `Pass` → `Fail` → `Pass` tuples three times. To do this we created three GitHub repositories for each project, each on a separate account and connected these accounts to Travis CI. We then created a branch at the commit associated with each of the failed builds. The branches are now pushed to each of the GitHub repositories, triggering three identical builds for each branch on Travis CI. To ensure that a build is triggered when a branch is pushed we removed the branch whitelist and blacklist sections from the Travis configuration file and removed the deploy and notification hooks from the configuration file to avoid disturbing the original developers.

If all three re-executions failed we considered the build to be non-flaky, and to be conservative we also considered three passes to be non-flaky, despite a pass being different from the original failure. Only re-executions that included at least two different results were considered flaky (e.g. failed, failed, passed, or errored, passed, failed). Unfortunately, in many cases all three re-executions errored. We found that this usually happened when dependencies were no longer available (e.g., Codehause [5]). These data points amounted to 32% of the re-executions we tried and were discarded.

3.6 Parsing Travis CI Logs

Travis CI does not make the names of tests available through its API, but rather stores detailed logs that contain test results. In order to determine how often build failures are caused by a single test failure (Section 4.4) and to find a number of valuable and costly individual tests to analyze (Section 4.5) we parsed the logs and extracted the names of failed tests.

[3] https://github.com/apache/pdfbox
[4] https://issues.apache.org/jira/browse/PDFBOX-2802
[5] http://www.codehaus.org/

This is unfortunately an error-prone process due to the variety of log formats, and we succeeded in accurately parsing the logs of only 40 projects. The results of sections that depend on knowing the outcome of individual test executions are therefore based on this subset of projects. The names of the 40 projects can be found in Appendix B.2.

Chapter 4

Results

In this chapter we answer the research questions posed in Chapter 1. The data used in our analyses were gathered from Travis CI, Github, and by rerunning builds on Travis CI; these data sources and methods of collection are described in detail in Chapter 3 .

4.1 Build States and Transitions

Our study centres around test failures in practice. To this end, we gathered real build results from Travis CI as described in Section 3.1 for the projects described in Section 3.2. Our final dataset contains 61 open source projects that executed a total of 106,738 builds on Travis CI. Table 4.1 shows the number of builds in each of the possible states. We can see that while the majority of builds pass (66.8%) there is also a large percentage (18.4%) of builds that fail, indicating that developers do push faulty code quite frequently. The 18.4% of builds that fail represent an upper bound; it is the largest percentage of pushed change sets that test suites could have found a fault in (i.e., the majority of builds pass and could therefore not have found any faults). It is also surprising to to see that 14.5% of builds are in the error state, almost as many as in the failed state. This indicates that compilation, configuration, and dependancy issues are common.

Another interesting aspect is how builds transition between states. These transitions are shown in Figure 4.1a. Cancelled and started build states are not included in the table because they are relatively uncommon and do not provide additional insight into test suite outcomes.

Table 4.1: Aggregated build results for 61 open source projects that use Travis CI and the state of the 106,738 builds they executed.

State	Builds	% of Builds
Pass	71,303	66.8
Fail	19,640	18.4
Error	15,437	14.5
Cancel	354	0.3
Started	4	0.0
Total	106,738	100.0

We also visualize the transitions between fail, error, and pass states in Figure 4.1b. From the figure it is clear that for each of the three states a project is most likely to remain in a state, rather than transition between states. This is clearest for the pass state; projects remain in this state for an average of 5.6 builds. Projects remain in fail or error states for only 2.9 and 2.6 builds on average [1]. These numbers demonstrate that developers of the projects we examined in our study pay attention to test results and fix issues in a timely manner.

4.2 What portion of build failures are non-deterministic?

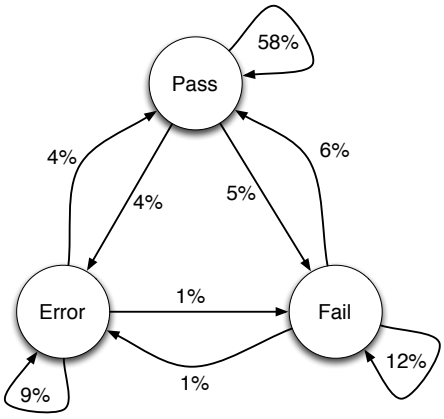
Research question 2 asked what proportion of test suite failures are non-deterministic. To answer this question we re-executed the failed build of 935 tuples according to the method described in Section 3.5. Of these, 12.8% did not execute consistently and can be considered flaky.

Flaky test failures can be a return on investment if the underlying problem is in product code and resolved, or a maintenance cost, if the problem lies in the test code. Because of their non-deterministic nature these failures are often difficult to reproduce and resolve which can lead developers to ignore the failures; this can be seen when the 'fixing' build of a tuple is made up of changes that are unrelated to the failing tests. We also found examples of multiple changes being pushed to Travis in order to debug an issue, indicating that the

[1] The median number of builds are one, one, and two for fail, error, and pass states respectively

Transition	Total	% of Transitions
Pass → Pass	58,330	57.6
Pass → Fail	5,253	5.2
Pass → Error	4,337	4.3
Fail → Pass	5,550	5.5
Fail → Fail	11,824	11.7
Fail → Error	1,354	1.3
Error → Pass	4,304	4.2
Error → Fail	1,448	1.4
Error → Error	8,929	8.8
Total	101,329	100.0

(a) Build state transitions for the 106,738 builds.



(b) Transitions between build states. Each transition is caused by one or more commits.

Figure 4.1: Build state transitions.

environment on Travis CI is likely different from the local development environment which can make it difficult to debug issues without pushing changes to Travis. Lastly, flaky tests that are hard to fix may also be disabled to keep them from interrupting developers. We found examples of all these categories of flaky failures during our manual inspection of tests in Section 4.6.

4.3 How often do failures represent a cost and how often do they represent a benefit?

A test failure is not necessarily a return on investment: it could also indicate a fault in the test suite itself that needs to be repaired. In this section, we describe how we determined what portion of build failures fall into each of the categories in Table 3.1 by linking build failures to the changes that caused them using the method described in Section 3.3, and then classifying the failures as a cost or benefit as described in Section 3.4.

We found 1,381 instances of `Pass` → `Fail` → `Pass` tuples, each instance contained three builds (by definition) and each build contained an average of 5.2 commits. Table 4.2 shows all the possible `Pass` → `Fail` → `Pass` tuples organized by the type of change that

Table 4.2: Distribution of tuples. The left-hand-side of → denotes the resources that changed to cause the test failure. The right-hand-side of → denotes the resources that were changed to resolve the test failure.

Fix Type	Resources Changed	Count	%
Code fixes	Code → Code	586	42.4
	Test → Code	37	2.7
	Code+Test → Code	170	12.3
	∗→ **Code**	**793**	**57.4**
Test fixes	Code → Test	80	5.8
	Test → Test	55	4.0
	Code+Test → Test	133	9.6
	∗→ **Test**	**268**	**19.4**
Mixed fixes	Code → Code+Test	120	8.7
	Test → Code+Test	22	1.6
	Code+Test → Code+Test	178	12.9
	∗→ **Code+Test**	**320**	**23.2**

fixed the build. Here we see that 57.4% of `Pass` → `Fail` → `Pass` tuples are fixed by production code changes only, in other words, the tests that failed must have found faults in the production code, thereby providing a return on investment. On the other hand, 19.4% of tuples transitioned back to a passing state by changing only test code; these test failures identified a fault in test code and therefore represent a maintenance cost. The remaining 23.2% of tuples were fixed by a change to both production and test code. To classify builds that were resolved with both production and test code changes we examined the proportion of changes that were made to the source code and the test code to fix the build. For these 320 tuples we observed that on average 43.9% of the edits were to test code, while the remaining 56.1% were to source code files. As we considered fixing Fail builds by changing the source code beneficial we classify 56.1% (13% of all changes) of the mixed builds as fixing defects and the remainder (10.2% of all changes) as test maintenance.

It is also interesting to note that a failure is most likely to be fixed by the same change type that caused it (e.g., 48.2% of failures caused by test-only changes are fixed by test-only changes). This makes intuitive sense: whatever the developer changed most recently is most likely to be the faulty code.

The middle column of Table 4.3 summarizes our results and answers our first research

Table 4.3: The middle column shows Fail builds resolved by fixing faults in product code and by maintaining the test suite. The last column adjusts the results for flakiness.

Change Type	Before Flakiness (%)	After Flakiness (%)
Fixing Faults	70.4	62.6
Maintaining Tests	29.6	24.6
Flakiness	—	12.8

question: what portion of test suite failures represent a return on investment? These results show that 29.6% of test suite failures do not provide a return on investment: they find faults in the test suite that need to be resolved. In the second column we have adjusted the results by placing flaky tuples in a separate category since these can be seen as a different type of maintenance cost. It is important to note that we are not claiming that fixing these failing test suites could not provide a benefit in the future, however, their maintenance cost must be considered when modelling the overall cost of automated testing.

4.4 How often do tests fail alone?

Research question 3 asked how often tests fail alone. Of all the tests that failed in projects where we could parse the test logs accurately (as described in Section 3.6), we found that only 6% of them ever fail alone. Another way to look at it is that 36% of failed builds have only one failed test. Preliminary manual investigation leads us to believe that tests that fail together typically fail because of the same root cause.

When multiple test failures expose the same fault one could see all but one of the failures as being redundant. This data could be used to help design more accurate test reduction strategies. Test reduction strategies typically use coverage overlap to determine whether a test is redundant or not but this alone can be inaccurate [9]. By using past failure data in addition to overlap, i.e., only considering a test redundant if it has both overlapping coverage and always failed with other tests in the past, the false-positive rate of test reduction strategies can be reduced. Any test that has failed alone in the past would therefore be considered non-redundant, regardless of coverage data. The impact that past failure data can have on test suite reduction approaches is however limited by the amount of past failure data available for each test.

It is, however, also possible that multiple failing tests may provide further insight into the underlying cause of failures. The successful execution of an integration test, for

instance, might depend on a large body of code making it difficult to locate the root cause of a failure. A unit test that fails with the integration test might narrow the search to a specific method thereby simplifying the investigation.

4.5 Test value within and across projects

In Section 4.3 we determined that 24.6% of the Travis CI build failures we studied are resolved through test suite maintenance. Research question 4 asks how the maintenance cost of tests differs between projects and whether some tests are much more valuable than others.

To answer this question we parsed the build logs of the failing builds in our tuples in order to gather the names of tests that failed (explained in Section 3.6). Next, we assigned each individual test a score: the number of code-only fixes minus the number of test-only fixes. A positive number tells us that the test found bugs in the code under test on more occasions than it required maintenance. A negative number indicates the opposite: the test required maintenance more often than it found bugs. This measure is imperfect, in that finding two bugs might be worth the cost of a test requiring maintenance five times (i.e., we did not take severity into account).

Figure 4.2 shows the average score for each project's tests. We elided `apache/pdfbox` for clarity (it has a score of 39.25) and projects where the sum of test-only and code-only fixes is smaller than five are removed to reduce clutter. The size of the bubbles indicates the portion of test failures that were resolved by a mixture of code and test changes. This portion can be seen as the portion of test fixes where there is some doubt whether the fix is a return on investment or a maintenance cost (in Section 4.3 we argued that 43.9% of these fixes represent a maintenance cost). On the x-axis we plot the sum of all test failures that were resolved by code or test changes alone on a logarithmic scale, this is an indication of the amount of data we are basing our analysis on.

From the figure we can see that, on average, the tests of nine projects require maintenance more often than they find bugs, it is therefore possible that these test suites add very little value, or even represent a loss for the projects. We also notice that the difference between code-only fixes and test-only fixes for most projects is small, ranging from -0.87 (`graylog2-server`) to 1.16 (`cloudify`) when excluding the clear outlier, `apache/pdfbox`, where the difference is 39.25. There are however individual tests that provide a large return on investment (24 tests have a score of 5 or higher) and also tests that may be costly to maintain (609 tests have a score of -1 and 4 tests a score of -2). In the next section we manually investigate a selection of these tests and their transitions into and out of failure.

4.6 How do the `Pass` → `Fail` → `Pass` transitions of tests with high and low ROI differ?

By looking at the difference between the number of times a test finds a bug and the number of times a test requires maintenance we were able to identify both tests that require maintenance more often than they find bugs as well as tests that find bugs more often than they require maintenance. We manually analyzed a selection of valuable tests in Section 4.6.1 and costly tests in Section 4.6.2. More details about the valuable and costly tests referenced in these sections are available in Table 4.4 and Table 4.5 respectively.

4.6.1 Valuable Tests

We first inspected the three most valuable tests (i.e., tests where the difference between code-only fixes and test-only fixes is the largest). All three tests are from `apache/pdfbox`, which has a score of 39.25, the highest of any project. The first (Test 1) experienced a code-only fix 86 times and a test-only fix 6 times. The test failed such a large number of times because it was dependent on test execution order and therefore did not find 86 bugs as it may appear at first glance but rather failed randomly. The issue is documented in a Jira bug [2], and the fix ensures that a directory the test needs exists before executing the suite. The test that failed the second most (Test 2) always fails with the first test, and for the same reason.

The third most valuable test (Test 3) is also from `apache/pdfbox` and is an example of a test where, because of the complexity of these projects, we are uncertain as to the connection between breaking and fixing changes. Because of the large number of failures (57 code-only, 2 test-only, and 3 code-and-test fixes) and the disconnect between breaking and fixing changes we suspect that it may be non-deterministic. Another explanation is that the method it tests `renderImage()`, is a part of core functionality that depends on a large amount of code and therefore breaks frequently for reasons that are not apparent to an outside observer. This test also fails with other tests 90.3% of the time.

We stopped our investigation of apache/pdfbox tests at this point as the first 19 most valuable tests are from this project and there is a gap of 23 points between these and the next most valuable tests, leading us to believe that a large amount of non-determinism was involved in all of these tests.

[2] https://issues.apache.org/jira/browse/PDFBOX-2802

Table 4.4: Manually inspected valuable tests as well as an example of a broken build and the change that fixed the test.

(1)	TestFontEmbedding.testCIDFontType2 Project: apache/pdfbox Example Break: https://travis-ci.org/apache/pdfbox/builds/62491790 Example Fix: https://travis-ci.org/apache/pdfbox/builds/62506331
(2)	TestFontEmbedding.testCIDFontType2Subset Project: apache/pdfbox Example Break: https://travis-ci.org/apache/pdfbox/builds/62491790 Example Fix: https://travis-ci.org/apache/pdfbox/builds/62506331
(3)	TestRendering.render Project: apache/pdfbox Example Break: https://travis-ci.org/apache/pdfbox/builds/29020886 Example Fix: https://travis-ci.org/apache/pdfbox/builds/29021184
(4)	org.apache.jackrabbit.oak.jcr.observation.ObservationTest.observationDispose[1] Project: apache/jackrabbit-oak Example Break: https://travis-ci.org/apache/jackrabbit-oak/builds/12831165 Example Fix: https://travis-ci.org/apache/jackrabbit-oak/builds/12868383
(5)	org.apache.jackrabbit.oak.jcr.security.user.UserImportTest.testImportGroupIntoUsersTree Project: apache/jackrabbit-oak Example Break: https://travis-ci.org/apache/jackrabbit-oak/builds/7116911 Example Fix: https://travis-ci.org/apache/jackrabbit-oak/builds/7116984

Next we looked at two tests that are not such extreme outliers. The first (Test 4) failed four times with code-only fixes. The first failure was due to a typo that resulted in 2125 test failures, this was corrected in the fix. The second failure was due to a change that slowed the test enough to result in a time out, the fix partially reverts the change to fix this. Similarly, the third and fourth failures are both time outs and neither are non-deterministic. The last failure is the only instance of this test failing alone and it was therefore absolutely necessary for catching a bug only once.

The second test we looked at (Test 5) failed three times with a code-only fix. The first failure found a real bug, in the second failure the test was one of the 2125 tests that failed due the typo we described above. The third failure found a real issue and the fix reverted the breaking changes.

4.6.2 Costly Tests

There are not as many tests with a negative score as there are with a positive score, and the largest negative scores are much smaller than the largest positive scores. As we saw in the previous section, the outliers on the positive side are mostly due to non-deterministic tests and the lack of such extreme outliers on the negative side may be due to developers making more changes to product code than test code. This means that whenever a flaky test fails it is more likely that the change which seems to 'fix' it is a product code change. There are however 613 tests (16.9% of all tests in tuples) that require maintenance more often than they find bugs. We inspected nine tests with negative scores to see whether they have anything in common and whether these failures could have been avoided. Four of the tests experienced test-only fixes twice. The remaining five experienced only one test-only failure and none of the tests experienced code-and-test or code-only fixes.

Two of the tests failed due to unintended interaction between tests. The first test (Test 6) depended on an `@after` method deleting the user with email address `JO@FOO.COM` from the database but failed because the user's email address was in lower case in the database. This test used to work since MySQL, by default, is not case sensitive but stopped working when the test was run on Postgres which is case sensitive by default. This failure could have been avoided had the developers not depended on the default behaviour of MySQL. The second test (Test 7) was fixed by re-initializing the context between tests. It is however interesting that the breaking change removed the re-initialization code and the fix simply replaces it. It is hard to know why it was removed in the first place.

Two of the tests were non-deterministic and from the commit messages we know that the developers were aware of this. The 'fixing' commit for the first test (Test 8) has the commit message "more logging to debug saveTask_shouldSaveTaskToTheDatabase", making it clear that the issue is not yet resolved. When the test fails again at a later stage the 'fix' seems to be unrelated to the failing test and we suspect that it does not resolve the issue. The second test (Test 9) has commit message "Try to fix flakiness of NaturalDateParser..Collections.sort() seems to be indeterministic..." indicating the developers are already aware that the test is non-deterministic. After the second failure of this test the developers disabled it.

Two of the tests (Test 10 and Test 11) we inspected were too difficult to analyze and we could not determine whether they are non-deterministic or not.

One test (Test 12) as well as the four others that failed with it are an example of new tests being added to the suite that fail on the first execution. They were fixed by changing a constant.

Of the last two tests the first is an example of a functional change resulting in test suite maintenance (Test 13) and the second is an accident in which changes were applied before they were ready (Test 14).

4.6.3 Discussion of manual inspection results

From the valuable tests we investigated we see that the most valuable tests are in fact non-deterministic tests that did not find bugs in the code under test and these failures can therefore be considered a cost rather than a benefit. Tests that experienced lower numbers of code-only fixes are more likely to have found real bugs and provided real value. Our second observation is the diversity in causes of negative test failures. These include tests that fail because of interaction between tests, non-deterministic tests, new tests failing immediately after creation, tests failing after a change in functionality as well as an accident where changes were merged before they were ready. This reveals that an array of techniques are most likely needed in order to significantly reduce maintenance costs. The prevalence of flaky tests, as well as the difficulty developers have with preventing and resolving these failures, stand out as major costs in automated testing. They hinder empirical studies of test results as it is hard to automatically distinguish between deterministic and non-deterministic failures.

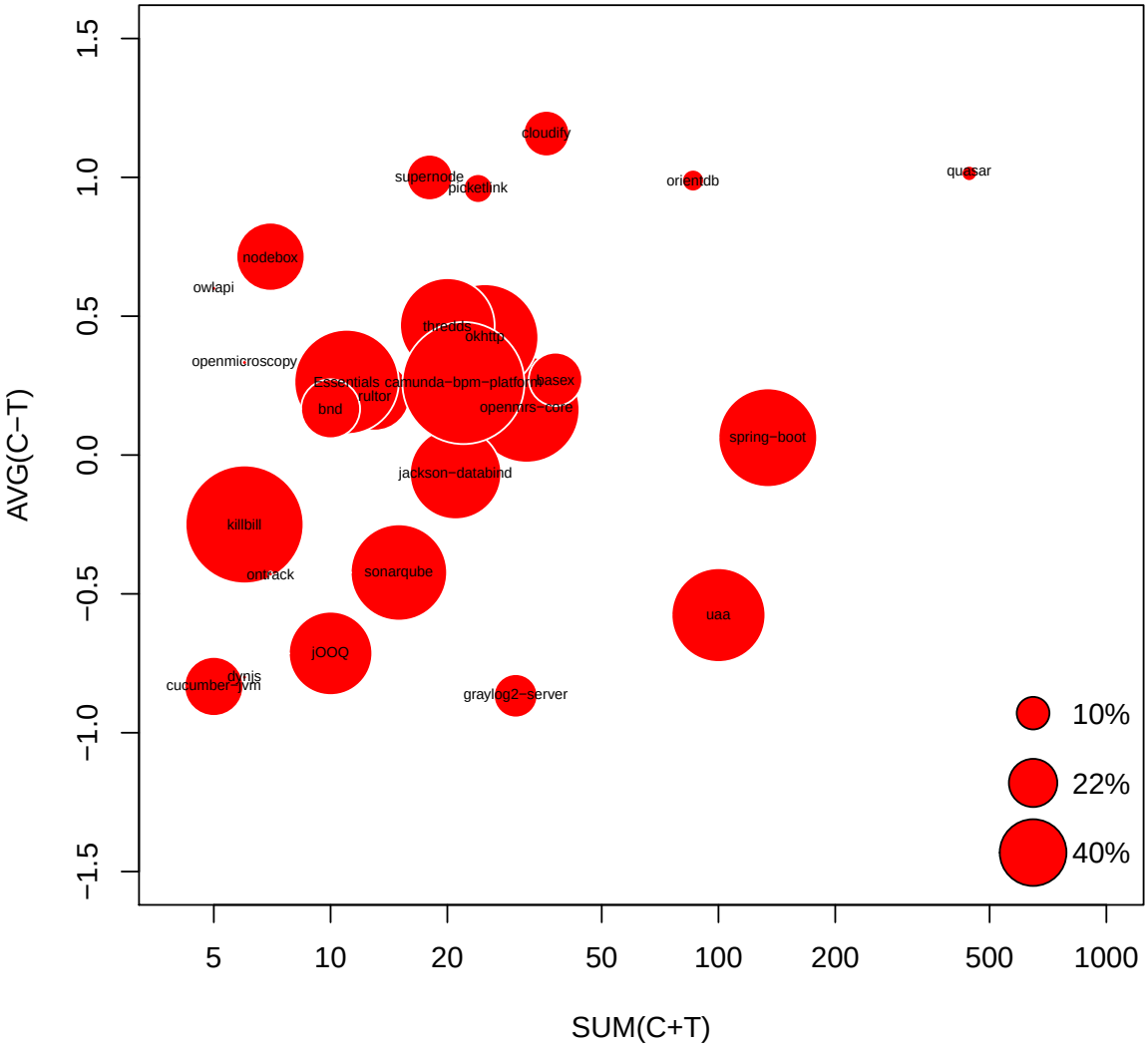


Figure 4.2: Average difference between code-only and test-only fixes. The radius of the bubbles represents the relative size of mixed fixes compared to all fixes (i.e., CT/(C+T+CT)). The sum of all code-only and test-only fixes for a project is plotted on the x-axis on a logarithmic scale to give a sense of the size of the data set. The y-axis shows the average difference between code-only fixes and test-only fixes i.e., AVG(code-fixes - test-fixes).

Table 4.5: Manually inspected costly tests as well as an example of a broken build and the change that fixed the test.

(6)	org.cloudfoundry.identity.uaa.scim.jdbc.JdbcScimUserProvisioningTests.canCreateUserWithoutGivenNameAndFamilyName Project: cloudfoundry/uaa Example Break: https://travis-ci.org/cloudfoundry/uaa/builds/40103431 Example Fix: https://travis-ci.org/cloudfoundry/uaa/builds/40114195
(7)	org.basex.test.query.expr.PackageAPITest.delete Project: BaseXdb/basex Example Break: https://travis-ci.org/BaseXdb/basex/builds/1093013 Example Fix: https://travis-ci.org/BaseXdb/basex/builds/1093360
(8)	org.openmrs.scheduler.SchedulerServiceTest.saveTask_shouldSaveTaskToTheDatabase Project: openmrs/openmrs-core Example Break: https://travis-ci.org/openmrs/openmrs-core/builds/31049916 Example Fix: https://travis-ci.org/openmrs/openmrs-core/builds/31125642
(9)	NaturalDateParserTest.testTemporalOrder Project: Graylog2/graylog2-server Example Break: https://travis-ci.org/Graylog2/graylog2-server/builds/42398852 Example Fix: https://travis-ci.org/Graylog2/graylog2-server/builds/42401387
(10)	org.jooq.example.NashornTest.testScripts Project: jOOQ/jOOQ Example Break: https://travis-ci.org/jOOQ/jOOQ/builds/41369551 Example Fix: https://travis-ci.org/jOOQ/jOOQ/builds/41373407
(11)	org.basex.test.query.FTIndexQueryTest.testFTTest Project: BaseXdb/basex Example Break: https://travis-ci.org/BaseXdb/basex/builds/1787079 Example Fix: https://travis-ci.org/BaseXdb/basex/builds/1792995
(12)	org.basex.test.query.func.FNClientTest.clientClose Project: BaseXdb/basex Example Break: https://travis-ci.org/BaseXdb/basex/builds/1434048 Example Fix: https://travis-ci.org/BaseXdb/basex/builds/1434164
(13)	org.basex.test.query.func.JavaFuncTest.staticMethod Project: BaseXdb/basex Example Break: https://travis-ci.org/BaseXdb/basex/builds/848008 Example Fix: https://travis-ci.org/BaseXdb/basex/builds/850290
(14)	org.basex.test.server.CommandTest Project: BaseXdb/basex Example Break: https://travis-ci.org/BaseXdb/basex/builds/759351 Example Fix: https://travis-ci.org/BaseXdb/basex/builds/759996

Chapter 5

Threats To Validity

There are several factors that may impact the validity of our work. We have divided them into three categories: Construct Validity, Internal Validity, and External Validity.

5.1 Construct Validity

In our study we classified build failures as either a maintenance cost or a return on investment by looking at the type of changes that resolved the failure. It is important to note that we do not assign a monetary value to test failures: it is possible that the value of finding a bug in product code far out weighs the cost of a bug in test code. The large number of bugs in test code does however represent an opportunity for improvement.

Our decision to study `Pass` → `Fail` → `Pass` tuples might have influenced our results, but as explained in Section 3.4 there are problems with any alternative we might have chosen.

To obtain the data on test flakiness described in Section 4.2 we re-executed each test three times. It is possible that re-executing the tests more than three times would have revealed more flaky tests.

5.2 Internal Validity

As with any software project, there are no guarantees that the code used to analyze the data is completely free of bugs. To mitigate this risk we wrote automated tests and performed

manual inspections. Another possible source of errors is the classification of files as product-code, test-code, or non-code. Although most projects follow the convention of placing tests in folders and files whose names contain the word 'test' there may be exceptions to this rule causing us to misclassify files.

Another threat is the list of file extensions (Appendix B.1) we used to identify code files as opposed to non-code files such as text or image files. This list was compiled by manually evaluating the most common file extensions from the projects we studied. As with any manual analysis there is a risk of human error.

5.3 External Validity

External validity refers to the generalizability of our results. We investigated test evolution and failures on the free open-source tier of Travis CI. It is possible that developers using other continuous integration services or the paid tier of Travis CI evolve their test suites differently. It is, for instance, possible that developers would fix flaky tests sooner if they were paying every time the tests ran.

The projects we studied are another threat to external validity. All of the projects are open source and therefore follow a distributed development model and may have different quality control processes and expectations than industrial projects. Most of the projects are written in the Java programming language and use JUnit for unit testing. It is therefore possible that some of our results are unique to Java and JUnit programming or the culture of the community that uses these technologies. We also eliminated a number of projects from our study due to unusual testing behaviour as explained in Section 3.2. The 61 projects we investigated are however diverse and one of the largest datasets studied in this area.

Chapter 6

Future Work

As future work we envision a tool that could simplify the study of test evolution both for researchers and developers. Currently, Travis CI provides a build history that makes shows how the build transitioned between states over time, but provides no way to easily see how individual tests transitioned between failed and passed states and what caused these transitions.

Such a tool would allow developers to identify tests that are expensive to maintain. The hope is that by listing the changes that led to a specific test requiring maintenance developers will be able to find a pattern in the causes. The test may, for instance, test the internal workings of a method rather than the end result, leading to unnecessary failures when implementation details change. By including a list of other tests that fail with the test being investigated one might also realize that two tests overlap and that it might be better to remove one of them. Finally, the test may need frequent maintenance because of constant requirements changes, indicating that it may be too early to use automated testing in this part of the application.

We believe that studying how test and product code co-evolve is not enough, but that what drives this evolution must be taken into account. A test suite has only done a small part of its job when all tests pass. It is when failing tests force the system under test to evolve that tests do their real work and when the system under test forces the test suite to evolve that they become a real burden. Studying this push and pull and helping developers to keep the performance of tests in mind on a day to day basis might provide new insights into when, what, how, and how much to test.

Chapter 7

Conclusion

In this thesis we studied the maintenance costs of automated test suites when used in a continuous integration environment. We found that 19.4% of build failures do not identify a bug in the source code but rather find a fault in the test suite itself and that 12.8% of build failures are non-deterministic. We also found that tests often fail together and that there are both costly and valuable tests. We hope that studying the difference between costly and valuable tests will allow researchers and developers to identify good practices that can reduce maintenance costs.

APPENDICES

Appendix A

Replication Package

A.1 Database Schema

Table Name: *repositories*	
Field	Type
id	int
slug	varchar(255)
last_build_id	int
last_build_number	int
last_build_state	varchar(255)
github_language	varchar(255)
contributors_count	int
github_language_github	varchar(255)
created_at	date

Table Name: *tests*	
Field	Type
id	int
job_id	int
name	text
state	varchar(255)
fix_type	varchar(255)

Table Name: *builds*	
Field	Type
id	int
repository_id	int
commit_id	int
number	int
pull_request	boolean
pull_request_title	varchar(255)
pull_request_number	int
config	text
state	varchar(255)
started_at	datetime
finished_at	datetime
duration	int
job_ids	text

Table Name: *rerun_builds*	
Field	Type
id	int
repository_id	int
build_id	int
rerun_commit_id	int
number	int
pull_request	boolean
pull_request_title	varchar(255)
pull_request_number	int
config	text
state	varchar(255)
started_at	datetime
finished_at	datetime
duration	int
rerun_job_ids	text

Table Name: *jobs*	
Field	Type
id	int
build_id	int
repository_id	int
commit_id	int
log_id	int
number	decimal
config	text
state	varchar(255)
started_at	datetime
finished_at	datetime
duration	int
allow_failure	boolean
annotation_ids	text
queue	text
expected_failed_tests_count	int
expected_errored_tests_count	int
found_test_result_line	boolean
expected_completed_tests_count	int
expected_skipped_tests_count	int

Table Name: *rerun_jobs*	
Field	Type
id	int
rerun_build_id	int
repository_id	int
rerun_commit_id	int
log_id	int
number	decimal
config	text
state	varchar(255)
started_at	datetime
finished_at	datetime
duration	int
allow_failure	boolean
annotation_ids	text
queue	text
expected_failed_tests_count	int
expected_errored_tests_count	int
found_test_result_line	boolean
expected_completed_tests_count	int
expected_skipped_tests_count	int

Table Name: ***commits***	
Field	Type
id	int
sha	varchar(255)
branch	varchar(255)
message	text
commited_at	datetime
author_name	varchar(255)
author_email	varchar(255)
commiter_name	varchar(255)
commiter_email	varchar(255)
compare_url	varchar(255)
in_repo	boolean

Table Name: ***rerun_commits***	
Field	Type
id	int
sha	varchar(255)
branch	varchar(255)
message	text
commited_at	datetime
author_name	varchar(255)
author_email	varchar(255)
commiter_name	varchar(255)
commiter_email	varchar(255)
compare_url	varchar(255)
in_repo	boolean

Appendix B

Other Tables

B.1 List of Code Extensions

A list of file extensions we considered to be 'code', as apposed to non-code files such as `.png` files. We compiled the list by going through the extensions that make up 90% of the files in our subject projects. After looking up the description of the file type we decided whether or not it qualifies as code.

Extensions of files considered 'code' files:

'.groovy', '.java', '.sh', '.rb', '.py', '.cc', '.c', '.cpp', '.js', '.coffee', '.scala', '.html', '.sql', '.jsp', '.jsx', '.xhtml', '.htm', '.frl', '.bat', '.hbs', '.obda', '.xbl', '.vm', '.erb', '.owl', '.lang', '.template', '.ttl', '.spec', '.h', '.xpl', '.go', '.m', '.pl'

B.2 Projects with Parsed Logs

We were able to extract test names from the logs of 40 of the 61 projects. The 40 projects are:

apache/jackrabbit-oak
apache/pdfbox
CloudifySource/cloudify
spring-projects/spring-boot
wordpress-mobile/WordPress-Android
square/okhttp
openmrs/openmrs-core
yegor256/rultor
camunda/camunda-bpm-platform
puniverse/quasar
Graylog2/graylog2-server
neophob/PixelController
lucmoreau/ProvToolbox
RoyalDev/RoyalCommands
Jasig/cas
timmolter/XChange
weld/core
SpoutDev/Spout
square/picasso
ansell/owlapi
killbill/killbill
openmicroscopy/openmicroscopy
SonarSource/sonarqube
Unidata/thredds
BaseXdb/basex
FasterXML/jackson-databind
essentials/Essentials
orientechnologies/orientdb
cloudfoundry/uaa
bitsofproof/supernode
bndtools/bnd
cucumber/cucumber-jvm
SeqWare/seqware
nodebox/nodebox
dynjs/dynjs
jOOQ/jOOQ
dcoraboeuf/ontrack
SonarSource/sonar-java
picketlink/picketlink
SpoutDev/Vanilla

References

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 142–151, 2014.

[2] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the European Software Engineering Conference held jointly with the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190, 2015.

[3] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 235–245, 2014.

[4] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418, 2009.

[5] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is this a bug or an obsolete test? In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 602–628, 2013.

[6] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 483–493, 2015.

[7] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: Empirical observations. *Advances in Software Engineering*, pages 4:1–4:13, 2010.

[8] Negar Koochakzadeh and Vahid Garousi. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering*, 2010, 2010.

[9] Negar Koochakzadeh, Vahid Garousi, and Frank Maurer. Test redundancy measurement based on coverage information: evaluations and lessons learned. In *2009 International Conference on Software Testing Verification and Validation*, pages 220–229. IEEE, 2009.

[10] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[11] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[12] Cosmin Marsavina, Daniela Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 195–204, 2014.

[13] Atif M Memon and Mary Lou Soffa. Regression testing of guis. *SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.

[14] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software evolution*, pages 173–202. Springer, 2008.

[15] Seyed Mehdi Nasehi and Frank Maurer. Unit tests as api usage examples. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[16] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 33:1–33:11, 2012.

[17] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 130–140, 2002.

[18] Arie Van Deursen. Program comprehension risks and opportunities in extreme programming. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 176–185. IEEE, 2001.

[19] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github.** updated version with corrections**. *arXiv preprint arXiv:1512.01862*, 2015.

[20] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 220–229, 2008.