

Change-Driven Testing



Sven Amann and Elmar Jürgens

Abstract Today, testers have to test ever larger amounts of software in ever smaller periods of time. This makes it infeasible to simply execute entire test suites for every change. Also it has become impractical—if it ever was—to manually ensure that the tests cover all changes. In response to this challenge, we propose Change-Driven Testing. Change-Driven Testing uses Test-Impact Analysis to automatically find the relevant tests for any given code change and sort them in a way that increases the chance of catching mistakes early on. This makes testing more efficient, catching over 90% of mistakes in only 2% testing time. Furthermore, Change-Driven Testing uses Test-Gap Analysis to automatically identify test gaps, i.e., code changes that lack tests. This enables us to make conscious decisions about where to direct our limited testing resource to improve our testing effectiveness and notifies us about where we are missing regression tests.

Keywords Software testing · Test automation · Test intelligence · Regression-test selection · Test prioritization · Test-resource management · Risk management

1 A Vicious Circle

Today, testers have to test ever larger amounts of software in ever smaller periods of time. This is not only because software systems grow ever larger and more complex, but also because development processes changed fundamentally. Ten years ago, software was commonly released at most once or twice a year and only after an extensive test period of the overall system. Today, we see consecutive feature-driven releases within a few months, weeks, or even days. To enable this, development happens on parallel feature branches, and testing, consequently, needs to happen on each such branch as well as on the overall system.

S. Amann · E. Jürgens
CQSE GmbH, München, Germany

© The Author(s) 2020
S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_1

In response to this fast-growing demand for testing, companies invest in test automation and continuous integration (CI) to speed up test execution. However, we increasingly see that even automated test suites run for multiple hours or even days, especially on larger systems. As a result, such long-running test suites are typically excluded from CI and executed only nightly, on weekends, or even less frequent. As a result, the time between the introduction of a mistake in the code and its detection grows. This has severe consequences:

- Large amounts of changes pile up between two consecutive test runs, such that it becomes difficult to identify which particular change contains the mistake.
- Developers get feedback from tests only long after they did the changes that introduced the mistake, again making it more difficult for them to identify the mistake.
- The effects of multiple mistakes may overlap, such that one mistake may only be detected after another is fixed.

To make things worse, test automation addresses only half the problem: While it improves the efficiency of test execution, it does nothing to ensure that the testing is effective. In practice, we see that about half of the changes may escape even rigorous testing processes [1, 2]. And as testers strive to make test suites more comprehensive by adding additional tests, they also add to the runtime of the suites, thus, jeopardizing the benefits of automated tests and CI.

So how can we break this vicious circle and make our testing processes efficient and effective at the same time? The answer is surprisingly simple: *We align our testing efforts with the changes.* With an increasing number of changes to be tested, it has become infeasible to simply execute all tests after every change and it has become impractical—if it ever was—to manually ensure that all changes are tested adequately. But instead of resolving to test only rarely, we should keep testing frequently with focus on the changes, i.e., the code that might contain new mistakes. We call the idea of aligning our testing with the changes in application code *Change-Driven Testing*. Change-Driven Testing identifies 90% of the mistakes that our entire test suite may find in only 2% of the suite’s runtime [3] and informs us about any change to the code that we did not test [1, 2].

2 Test Intelligence

To achieve high-quality testing, we commonly need to answer questions such as which test we need to run, what else we need to test, or whether our test suite contains redundant tests. Since it is difficult to correctly answer such questions manually, our goal is to automatically answer them using existing data from the software development process. This approach is similar to the approach of Business Intelligence, which analyzes readily available data to gain insights on business processes. Therefore, we refer to our approach as *Test Intelligence*. Figure 1 illustrates it.

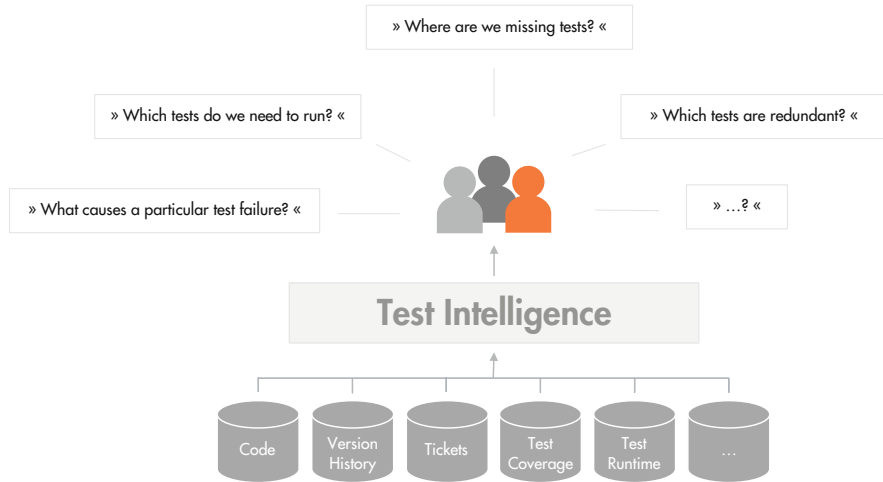


Fig. 1 Test intelligence: combining readily available data from various data sources in the software development process to automatically answer questions about testing

The questions we want to answer through Test Intelligence concern both our tests and the code changes under test. Therefore, we collect data about both the tests and the changes. Much of this data is already available in the development environment and need only be extracted for our purposes.

To communicate the extracted data to testers, developers, and managers alike, we use *treemaps*. Figure 2a shows such a treemap that represents the code of a UI component of the software system *Teamscale*. Each box on the map represents a single method in the code. The size of the box is proportional to the size of that method in lines of code. We color the boxes to highlight particular properties of the respective code.

2.1 Version-Control Systems

Version-control systems (VCS), such as `git` or TFS, are a de facto standard in today’s software development. Generally speaking, a VCS keeps a chronological history of code changes committed by developers and helps them coordinate their changes. Within the VCS, changes may be organized in branches, where a *branch* is an isolated line of changes that is based on a particular version of the code and may be merged into a later version of the code. If developers use a dedicated branch for the implementation of a particular feature, we call it a *feature branch*.

From the change history in a version-control system, we may extract the list of changes since any given baseline, be it a particular release, the last test run, the start of a feature branch, or any other point in time. Figure 2b shows code changes on a treemap. Methods that were added by one of these changes are highlighted in

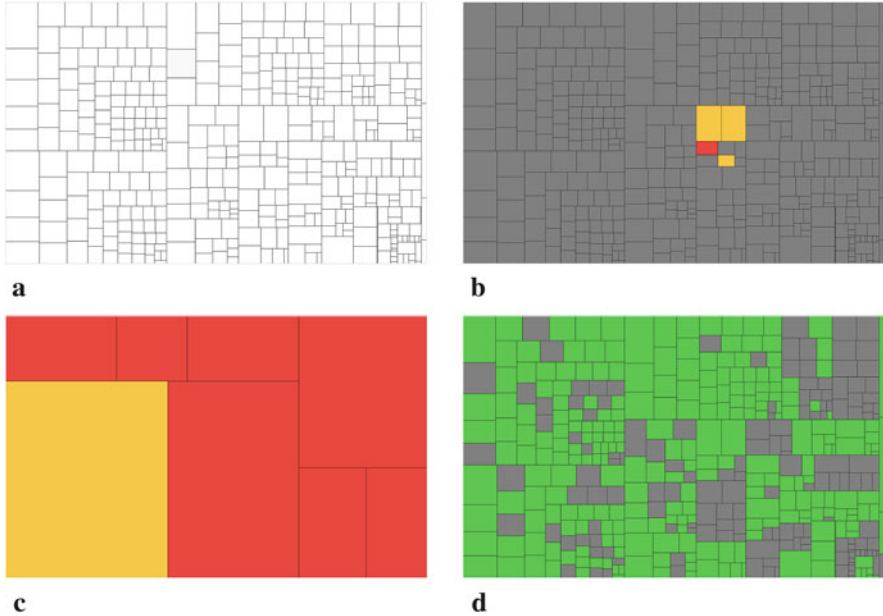


Fig. 2 Treemaps that show data about the software system Teamscale from the version-control system, the ticket system, and profiling test execution. (a) Source code as a Treemap. Each box represents one method. The size of the box is proportional to the lines of code in that method. (b) Code changes since a fixed baseline. New methods are red, changed methods are yellow, and methods that remain unchanged are gray. (c) Code changes for ticket TS-15717. New methods are red and changed methods are yellow. (d) Test coverage on method-level. Methods that have been executed are green, all other methods are gray

red, methods that were changed are highlighted in yellow, and methods that remain unchanged are highlighted in gray.

2.2 Ticket Systems

Ticket systems, such as Jira or GitHub Issues, are used in most software projects to keep track of change requests, such as bug reports, feature requests, or maintenance tasks. Such systems allow the development team to manage each request as a ticket, which usually has a unique ID, an assignee, and a status, among other metadata.

It is a widely used practice that developers annotate changes in the version-control system with the ID of the ticket(s) that motivated the changes, usually by adding the ID to the commit message that describes the respective change. Using these annotations of the code changes and the metadata from the ticket system, we

can group all changes that belong to the same ticket. This allows us to focus on only the changes motivated by a particular ticket.

Figure 2c shows the changes related to a single ticket on a treemap. Methods that were added by a change are highlighted in red and methods that were changed are highlighted in yellow.

2.3 Profilers

Profilers, such as `JACOBO` or `gCOV`, record which parts of the application code are executed, i.e., they record *code coverage*. Depending on the technology in use, profiling approaches range from instrumenting the target code, over attaching a profiler to a virtual machine, to using hardware profilers. Regardless, profiling is always possible.

Different profilers record coverage at different granularity, e.g., they record which methods, statements, or branches get executed. With most profilers, a finer granularity results in a larger performance penalty. Recording coverage on method level is, in our experience, always feasible with an acceptable overhead.

When a profiler records the coverage produced by a test, we speak of *test coverage*. Recording test coverage is a widely used practice for automated tests in CI and uncommon in other types of test environments. Technically, however, we may profile any type of execution, be it through a unit test or an end-to-end test, automated or manual. Thus, using a profiler, we may obtain the coverage of each test in our entire test suite.

Existing profilers typically aggregate the coverage of all tests executed in a single test run, because they work independently of the test controller and simply trace execution from start to end of the run. However, conceptually, we may also record *test-wise coverage*, i.e., separate coverage for each test case. When we record test-wise coverage, we may also trivially record each test's individual runtime.

Figure 2d shows the aggregated coverage of a test suite on a treemap. Methods that were executed are highlighted in green and methods that were not executed are highlighted in gray.

3 Change-Driven Testing

Change-Driven Testing is one particular instance of Test Intelligence that makes testing both more efficient and effective. Figure 3 depicts the idea. The key insight behind Change-Driven Testing is that existing tests will only fail if new mistakes are introduced,¹ and that new mistakes can only be introduced through changes. Consequently, when considering the (possibly buggy) changes between

¹Leaving aside causes for sporadic test failures, such as flaky tests or interaction with third-party systems, which typically indicate a problem with the test setup rather than with the system under test.

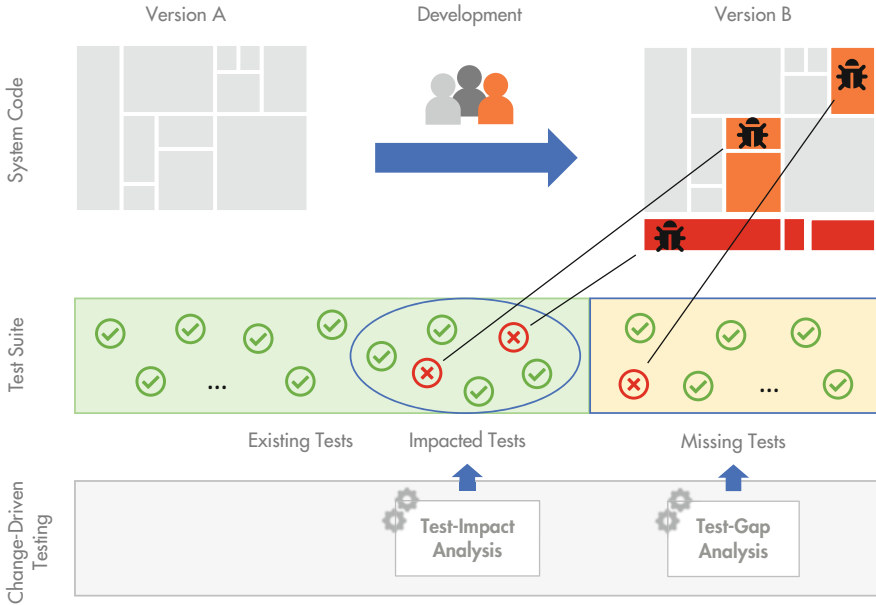


Fig. 3 Change-driven testing: testing efficiently by running only the tests impacted by changes. Testing effectively by testing all changes

two versions of our system, e.g., two consecutive releases of the system before and after implementing a particular feature, we proceed in two phases:

1. To make our testing more *efficient*, we test just the changes, excluding all other parts of the system. We automatically identify the existing tests that are relevant to the changes, i.e., the impacted tests, through a Test-Impact Analysis.
2. To make our testing more *effective*, we ensure that we test all changes. To guide us towards this goal and to verify whether we achieved it, we automatically identify changes that lack tests through a Test-Gap Analysis.

Both analyses are only interested in changes that require testing. Therefore, we use static code analyses to identify *relevant changes*. We consider a change relevant if it modifies the *behavior* of the code and, thus, may contain mistakes that later cause errors. Consequently, we filter out changes that correspond to refactorings, such as changes to documentation and renaming or moving of methods or variables.

3.1 Test-Impact Analysis

Figure 4 depicts the process of the Test-Impact Analysis (TIA). It combines relevant code changes (see Sect. 2.1) with test runtimes and test-wise coverage (see Sect. 2.3)

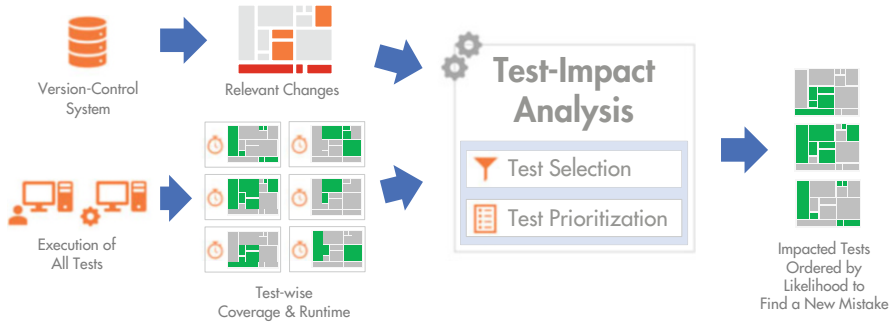


Fig. 4 Process of the Test-Impact Analysis (TIA). Given a set of changes, TIA selects the impacted tests and orders them by their likelihood to find a new mistake in the changes

to compute a subset of the entire test suite that identifies as many mistakes as early as possible. The process consists of two steps:

1. In the *Test Selection* step, TIA selects the *impacted tests*, i.e., all tests that execute any changed method, according to the recorded coverage. It also includes all tests that were added or modified by the change, because it cannot know which parts of the code those tests cover.
2. In the *Test Prioritization* step, TIA orders the impacted tests such that all changes are covered as quickly as possible, to find new mistakes as early as possible. Since computing the optimal ordering of the tests is infeasible, TIA uses a greedy heuristic: It selects that test next, which covers the most additional changed code per execution time.

In a study with twelve software systems [3] we found that the impacted tests selected by TIA find 99.3% of all randomly inserted mistakes that the entire test suite could find. The impacted tests found more than 90% of those mistakes in all study systems and even 100% in seven of them.

In a second study with over 100 different systems [4] we found that TIA identifies on average over 90% of the mistakes that the entire test suite identifies in only 2% of the execution time of the entire suite. Using TIA is especially beneficial for small and local changes, where test selection results in a small set of tests. This perfectly suits our need to quickly test many incoming changes, which are usually small compared to the code of the entire system.

3.2 Test-Gap Analysis

Figure 5 depicts the process of the Test-Gap Analysis (TGA). It matches relevant code changes (see Sect. 2.1) with the aggregated test coverage (see Sect. 2.3) to identify those changes that were not covered by any test. We call these changes *test gaps*.

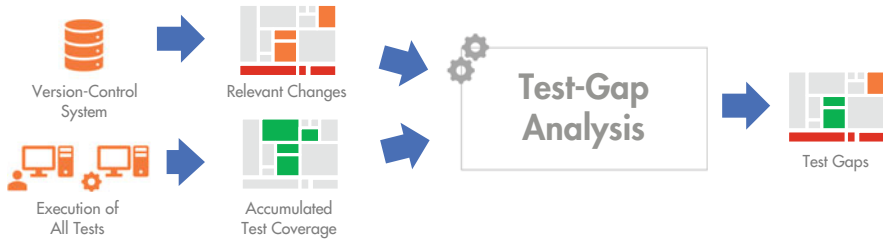


Fig. 5 Process of the Test-Gap Analysis (TGA). From a given set of changes TGA identifies those changes that were not yet tested, i.e., test gaps

To determine test gaps, TGA considers the chronological order of changes and test runs: New changes invalidate any previous test coverage of the changed code and open new test gaps. With subsequent testing, new coverage is recorded, which closes respective test gaps on earlier changes. Consequently, TGA can give us an update on our test gaps after every test run and every code change. And since the analysis works incrementally, it computes the update in a matter of seconds, even for very large systems.

In a case study on a large industrial software system [1] TGA revealed that more than 55% of the code changes in two consecutive releases remained untested. In retrospect, we traced over 70% of the reported field bugs back to untested code.

In a second case study on another industrial software system [5] TGA found 110 test gaps in the changes from 54 tickets, which corresponds to 21% of the 511 change methods. Provided with the data, developers found 35% of these gaps worth testing. Interestingly, they also found that 49% of the gaps resulted from cleanup work that was not part of the ticket description and, thus, remained untested even after the change requested by the ticket was thoroughly tested.

3.3 Limitations

To make best use of the analyses, it is important to be aware of their limitations. One limitation of both TIA and TGA is changes on the configuration or data level. Since such changes are not reflected in the code, they remain hidden from the analyses. Consequently, TIA cannot adequately select impacted test and TGA cannot show impacted parts of the code as untested.

A related limitation of TIA comes from the use of indirect calls. For example, if a test executes all classes with a certain annotation, TIA typically does not select this test when the annotation is added to another class, because the test's historical impact did not include that class and the test itself did not change on the code level.

TIA assumes that the coverage of test cases is stable, i.e., that executing the same test twice results in the same coverage. If this is not the case, e.g., because manual test steps are performed differently with every execution, TIA cannot properly

capture the impact of the test and, therefore, cannot adequately select impacted tests. In practice, even the coverage of automated tests may vary between runs, e.g., due to garbage collection. Though the effects are typically small, this means that TIA cannot guarantee that the impacted tests always find all the mistakes that the entire test suite may find. To the best of our knowledge, there is no test-selection strategy that gives such a guarantee under these conditions. Therefore, we recommend to regularly execute the entire test suite to ensure that nothing slipped through and to keep the data about test runtimes and coverage up to date. This is the same as using a traditional regular (e.g., nightly) testing strategy, except that we now have the additional fast feedback from the immediate testing using TIA, which already captures the majority of mistakes.

Another limitation of TGA is that it does not consider how thoroughly a change was tested, but only whether the methods containing the change were at all executed in a test. Like any other testing approach, TGA cannot prove the absence of mistakes in the code. However, by revealing changes that have not been tested at all, it identifies changes for which we are certain that no mistakes can have been found. Such untested changes are five times more likely to contain mistakes [2]. In our experience, TGA usually reveals substantial test gaps and, thereby, enables significant improvements of the testing process.

4 Using Change-Driven Testing

To illustrate the use of Change-Driven Testing, we follow the development of a feature of our own software product *Teamscale*. *Teamscale* is a platform that assists software development teams in the analysis, monitoring, and optimization of code and test quality. For this purpose, it integrates with various other development tools, such as version-control systems, build servers, and ticket systems. Its web frontend provides an administration perspective that allows to manage credentials for such external systems. Figure 6 shows feature request TS-15717 that asked to add the possibility to delete such external credentials.

Before starting to work on TS-15717, our developer creates a feature branch to work on. Then she implements the relatively small feature, writes a new test, and commits the changes as


TS-15717: Support deletion of external credentials.


The treemap in Fig. 2c shows all of these code changes.

4.1 Testing with TIA

Triggered by the commit, our CI environment starts building *Teamscale* and testing the change. In the testing stage, the CI queries TIA to learn which tests to execute. To compute its response, TIA uses the test-wise runtime and coverage of

Done **Issue 15717 - Enable deleting account credentials**

Creator:  Elmar Jürgens (on Dec 15 2018 13:53) Updated Dec 15 2018 13:53

Assignee:  Florian Dreier

Type	Priority	Fix Version	Component	QA-Contact
Feature	Normal	Teamscale 4.6	Web Interface	khater

Description

It would be useful for an admin to know which TS projects are configured to use a particular credential. Especially to locate unused credentials that can be deleted. This could be shown as a short list next to the credential name, similar to how we do this for the analysis profiles.

Fig. 6 Feature request TS-15717: enable deleting account credentials

each of the roughly 6.5k tests (including unit tests, integration tests, system tests, and UI tests) in our test suite. To obtain this data, we augmented the CI environment with profilers that record test-wise coverage in both the JavaScript code of the frontend and the Java code of the backend and ran the entire test suite once, which took about 45 min. Based on this data, TIA now determines a list of six tests impacted by the changes: Five regression tests covering parts of the changed code and the new test that came with the changes. The entire CI run with these impacted tests takes about 1.5 min, saving us over 96% runtime.

Thanks to TIA, only 1.5 min after committing her changes, our developer learns that the new test (ranked second by TIA) fails. With her changes still fresh in her mind, she investigates the problem and fixes it in about 10 min, committing the new changes as

TS-15717: Fix deletion of external credentials.

Since the fix is very local, TIA selects only a single impacted test, namely, the test that previously failed. Therefore, the second CI run takes only about 45 s in total. This time all tests pass.

Overall, two consecutive CI runs using TIA plus correcting the mistake in between took less time than one execution of our full test suite. Figure 7 illustrates this improvement.

4.2 Testing with TGA

To ensure that all her changes are properly tested, our developer next looks at the test-gap treemap for her changes in the context of TS-15717. Figure 8 shows how TGA displays the test coverage recorded in the two previous CI runs. The treemap shows what we call the *ticket coverage* of TS-15717: most of the code changes

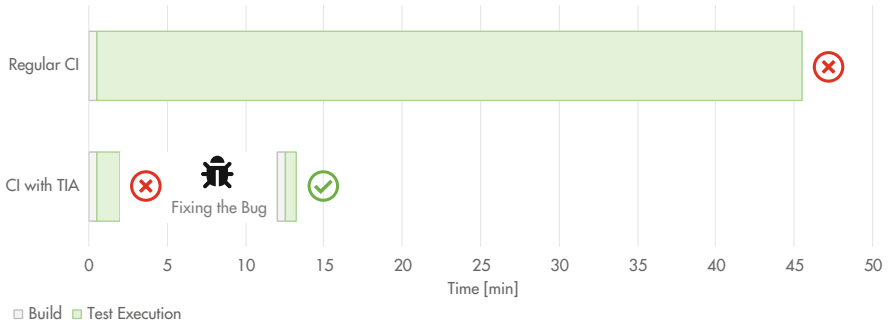


Fig. 7 The gain from using test-impact analysis. Two consecutive CI runs using TIA plus correcting a mistake take less time than one CI run with Teamscale’s full test suite

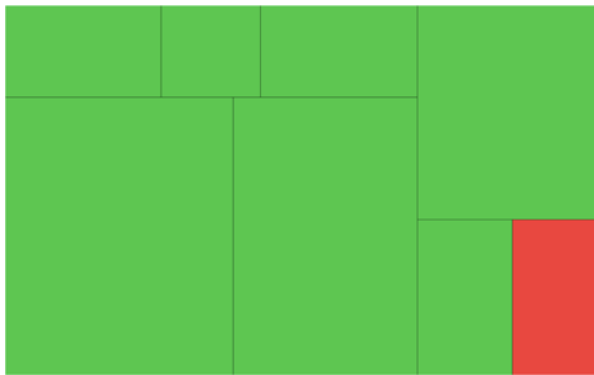


Fig. 8 Ticket coverage for TS-15717 after executing the impacted tests identified by TIA. TGA reveals one remaining test gap

were tested (the green rectangles), but one test gap remains (the red rectangle in the lower right).

To decide whether the remaining test gap is worth closing, our developer drills down from the treemap into the code. She discovers that the untested code is responsible for requesting an additional confirmation, when a user attempts to delete credentials that are still used by Teamscale. Since this code is relatively simple and unlikely to ever change, she decides to test it once manually.

Our developer starts the development version of Teamscale from her local machine, using a manual-test startup script that we maintain with our code. She opens the system under test in a browser, navigates to the administrative perspective, and checks whether the additional confirmation is indeed requested. It is, and so she shuts down the system under test, which causes the startup script to automatically provide the recorded coverage for TGA. As a result, all changes for TS-15717 were now tested, verifiably documented by an all-green test-gap treemap.

Since our developer opted for a manual exploratory test, there is no regression test for this particular functionality. However, since TGA is aware of the chronological order of changes and test coverage, it will again report a test gap should the functionality ever change in the future. Thanks to this safety net, it is reasonable to opt for a quick manual check instead of writing an automated UI test or a manual test for code that is unlikely to ever change again.

4.3 Closing the Loop

At this point, our developer is satisfied with her changes and sends them to one of her peers for code review. Once she and the reviewer agree that the changes are fine, he merges the feature branch. In response, our CI environment runs our entire test suite. This ensures that the main product line is error free, even if TIA should have mistakenly excluded a relevant test, and also records coverage and test execution times to keep our data up to date. Note that the vast majority of CI runs still benefits from TIA, since merging feature branches happens much less frequently than committing changes to feature branches.

Before each `Teamscale` release (as of this writing, every 6 weeks) a test architect inspects all remaining test gaps on changes since the last release across the entire system. This provides us with a second quality gate, to ensure that no critical functionality accidentally slipped through testing. In this process, the architect uses the same data that was used in the development process of the features, but on a treemap that represents the entire code instead of only the code changes for an individual feature. Figure 9 shows a section of this global test-gap treemap, representing one of `Teamscale`'s UI components.

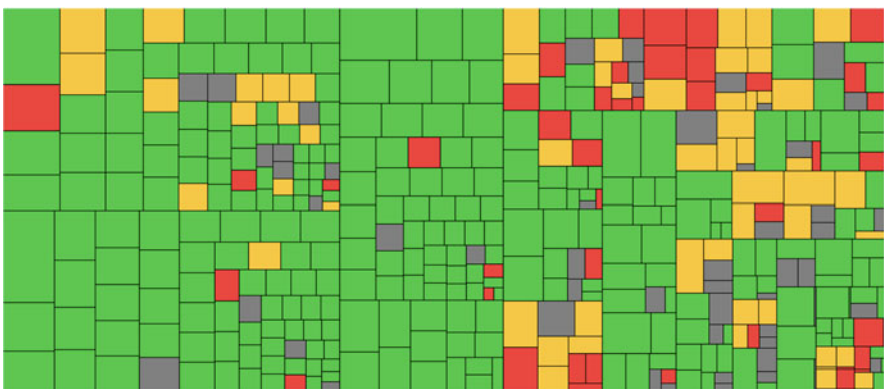


Fig. 9 Test-gap Treemap for a UI component of `Teamscale`. A global analysis of remaining test gaps serves as an additional quality gate before a release

5 Adapting Change-Driven Testing

In practice, we encounter very different testing setups and strategies, depending on the concrete requirements and the history of the respective projects. Consequently, the implementation of Change-Driven Testing should be adjusted to deliver the most value given the project's parameters. We subsequently discuss some aspects that we encounter repeatedly, without aiming for an exhaustive list.

- Both Test-Impact Analysis and Test-Gap Analysis may consider test coverage from all testing stages as well as from a combination of both automated and manual tests. The type of test literally makes no difference for the benefits of Test-Gap Analysis. Test-Impact Analysis, on the other hand, is especially beneficial if tests are time consuming, i.e., if excluding tests saves significant time, and if testing time is limited, because test prioritization makes it likely that we catch mistakes early, even if we cannot afford to run all impacted tests.
- If development mostly happens on feature branches, a sensible strategy is to use Test-Impact Analysis for testing changes and ticket coverage to avoid test gaps on these branches. In addition, to ensure no mistakes slip through testing, the full test suite should be executed upon merge of a feature branch. If, on the other hand, development happens on a main branch, we may use Test-Impact Analysis to test individual changes and run the full test suite periodically or before a release.
- It is always possible to combine Test-Impact Analysis with other test-selection strategies, e.g., if some tests for highly critical functionality should always run. We then simply run the union of the impacted tests selected by TIA and the tests identified by any complementary strategy.
- We found that it is most efficient to investigate test gaps using ticket coverage, because the ticket provides us with additional context when analyzing the gaps. However, even if TGA cannot map code changes to tickets, it can reveal test gaps for the system as a whole. Such a system-wide TGA is valuable on its own as much as in addition to ticket coverage, as a second-level quality gate.
- In many projects, the people analyzing test gaps are not necessarily the developers who wrote the code changes, but testers or architects. Such a separation of work sometimes makes the interpretation of test gaps more difficult, because the analysts may be unaware of possible reasons for a particular change or test gap. In such cases, the data from the version-control system again proves helpful, because it names the developer responsible for any change, telling the analysts who to talk to.
- While it is the theoretical ideal, it is never a goal in itself to reach an all-green treemap, i.e., 100% test coverage. In many situations, leaving test gaps is quite reasonable, e.g., if the gap is on code that prepares future functionality, but that it not yet live or if it is on code that is only rarely used internally, such that testing resources are better invested elsewhere. In the end, the testing process is always subject to tradeoffs and prioritization. TGA enables us to make conscious decisions about where to direct our limited resources.

6 Conclusion

Today, testers have to test ever larger amounts of software in ever smaller periods of time. This makes it infeasible to simply execute even fully automated test suites in their entirety for every change. Also it has become impractical—if it ever was—to manually ensure that the tests cover all changes. Therefore, we need to rethink our testing strategies to become both more efficient and effective.

In this chapter, we introduced Change-Driven Testing. In Change-Driven Testing, we analyze existing data from the software development process to automatically answer questions that drive our testing. We use Test-Impact Analysis to automatically find the impacted tests for any given code change and sort them in a way that increases the chance of catching mistakes early on. This makes testing more efficient, catching over 90% of mistakes in only 2% testing time. We use Test-Gap Analysis to automatically identify test gaps, i.e., code changes that lack testing. This enables us to make conscious decisions about where to direct our limited testing resource to improve our testing effectiveness.

References

1. Eder, S., Hauptmann, B., Junker, M., Juergens, E., Vaas, R., Prommer, K.H.: Did we test our changes? Assessing alignment between tests and development in practice. In: Proceedings of the Eighth International Workshop on Automation of Software Test (AST'13) (2013)
2. Juergens, E., Pagano, D.: Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice. Whitepaper, CQSE GmbH (2016)
3. Juergens, E., Pagano, D., Goeb, A.: Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites. Whitepaper, CQSE GmbH (2018)
4. Rott, J.: Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis. Master's thesis, Technische Universität München (2019)
5. Rott, J., Niedermayr, R., Juergens, E., Pagano, D.: Ticket coverage: putting test coverage into context. In: Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSOM'17) (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

