# Test Intelligence: How Modern Analyses and Visualizations in Teamscale Support Software Testing

Jakob Rott*
CQSE GmbH, Munich

**ABSTRACT**

Software development, as well as software testing, of a system spawns lots of different artifacts of that many remain unused in further engineering of the system. In practice, we see a possible reason in the lack of knowledge how these artifacts can be processed to profit the most. This is a problem as additional information can improve software testing by making it more effective and efficient. If unused, as a consequence, the testing process is suboptimal, either slow or expensive and might, in a worse case, also result in a bad overall system quality.

In this paper, we give an overview about how modern analyses and visualizations enable the access to processed artifacts that support software testing and make benefits clear to the engineering and management stakeholders of a system. We focus on analyses in the tool *Teamscale* which is an application for software quality evaluation. For each analysis presented, we state which problem it addresses and in which context it can be used.

With this summarized presentation, we equip the reader with valuable analyses that are in the right scope for stakeholders in software testing. We present how test data can be used in a continuous feedback process by showcasing the analysis visualizations. Many of them are already successfully employed in the software engineering process of companies from a variety of industry sectors.

**Index Terms:** Test Intelligence—Artifacts of Software Engineering and Testing—Visualization Techniques; Effective Software Testing—Test Gap Analysis; Efficient Software Testing—Test Selection and Prioritization—Test Impact Analysis / Pareto Testing

## 1 INTRODUCTION

Successful software systems grow. This holds true for applications that are developed for all industrial sectors and comes with the difficulty that large systems require extensive and thus costly testing.

Academia makes steady progress in their suggestions on how to improve and optimize software testing processes in different aspects. However, when considering the recent study of Hynninen et al. (2018) who report on industry standards in software testing it seems that in many testing environments fine-grained optimization is not yet an option as for several basic testing tasks no tool support is currently established at all on which one could build upon. [5]

If no tool is used, artifacts like code coverage of test runs, change information from the version control system, or tickets from issue trackers can most likely not be respected in further testing plans. This is coherent to what the authors observe in their regular work as quality engineers in diverse industry projects.

Consequently, test managers often direct the available resources for testing based on their experience (*subjective*) and do not put artifacts from past test runs and other data sources into consideration which would allow a more *objective* source of decision-taking. The reason for this might be that interpretation of the large amount of raw data is not possible but needs extensive preprocessing. Also suitable tool support and benefits might be unknown. As a consequence, several problems and questions remain:

- Even with parallelization, tests run too long to finish in a reasonable time frame. [1]
- After a testing phase—no matter whether for a single ticket or a large new release—it is unclear whether the executed tests ran over all changed code regions. In uncovered regions no faults can have been detected and research has shown that a substantial number of changes reaches production untested. [2]
- A large part of the testing budget is used to test code areas in which bugs are more unlikely than in other areas.
- Given a time budget for testing, a useful subset and execution order of test cases is unknown. [11]
- Test feedback reaches developers too late. [1]
- Are requirements, code and tests still in sync after one of them changes? [12]
- Which code change led to a test failure? [11]

In research, test analyses were proposed that can help to answer the mentioned questions. In industry, a modern tool like *Teamscale* [4] that implements these approaches offers in addition meaningful visualizations. The latter play an important role when it comes to making the use of modern approaches widespread as they clearly show the relations of different artifacts and highlight the benefits of using analysis results to all stakeholders.

Our contribution is an overview about different analyses and their visualization offered by *Teamscale*. To the best of our knowledge so far there is no publication that covers such a synopsis for any tool. The visualizations in this paper were not generated ad hoc but stem from the analyses of different software systems performed in the past years by professional quality engineers employed at *CQSE* and are approved to be reused in this context (see ack., p. 6). All involved creators lead or accompany the quality control process of diverse software development projects and the examples in this work were carefully chosen. The paper sums up for which *purpose* to use which analysis **[Purp.]**, their required *input data* **[Inp.]**, the resulting *benefit* **[Benef.]** and which roles they target **[Targ.]**. If present, we enrich the presentation with experiences from their application in industry and refer to evaluation studies.

The analyses relate to the research areas *Change-Driven Testing* (test process oriented towards changes, e.g., test case selection, prioritization and coverage gap analysis) and *Test Intelligence* (usage of available artifacts from software development and testing). [1]

The remainder of this work is organized as follows. In Section 2, we present *Test Gap* analyses that identify recent changes that remained uncovered in the tests. Section 3 discusses an integrated analysis of system requirements and test results into a *verification matrix*. Section 4 describes three analyses that cover *test case selection* and *test case prioritization*. An analysis to highlight changes that were done since a preceding test run in order to find the *cause of test failures* is presented in Section 5. In Section 6, we report on visualizations that reveal *interrelations* of testing data and statically calculated measures. Finally, Section 7 concludes the paper with a short statement regarding these modern techniques and their general application in testing and possible future work.

*e-mail: rott@cqse.eu

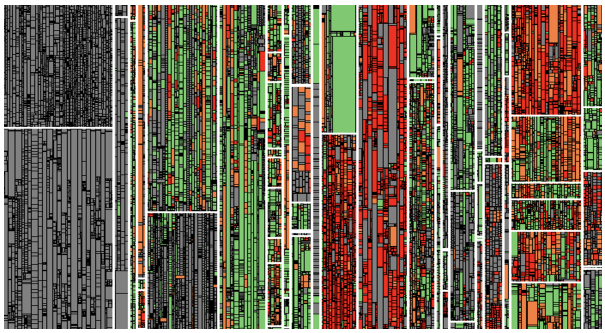

Figure 1: The treemap shows a *system-wide Test Gap analysis* (TGA) and gives an overview how thoroughly changes within a timespan (in this case, from release *x-1* to *x*) are covered by tests. Each rectangle represents a method in the source code. Its size corresponds to the length of the method (SLOC). Grey methods remained unchanged in the codebase since the last release. Defects would probably have been recognized since the code is regularly executed in a productive environment. Colored methods have been changed (orange) or added (red) since the last release but were not tested since the last modification. The green ones have either been changed or added and were executed by a test after the latest modification.
The shown treemap was created only a few days before the planned release. It revealed several larger untested modules (white borders). Based on this TGA, it was decided to shift the release date to enhance testing, since the Test Gaps were considered too grave.

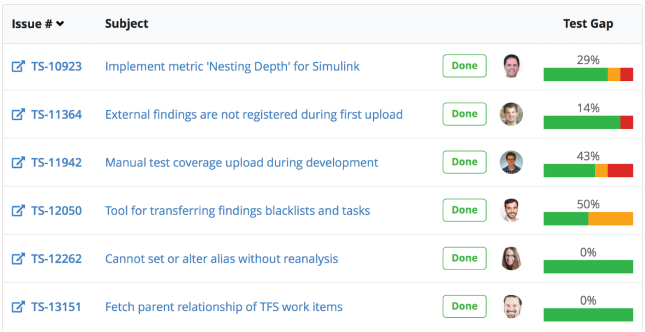


Figure 2: Bar charts on the right side indicate the *Test Gap for single issues*. For a list of issues this allows a quick overview which ones are still lacking method coverage significantly. In Teamscale this list can be filtered by specifying issue queries (e.g., tickets assigned to me, tickets belonging to the current sprint).

## 2 Test Gaps

Research has shown that the probability of bugs in code regions that have been changed since the last release but not tested afterwards is five times higher than in other areas of the codebase [2]. [Purp.] To reveal untested changes, *Test Gap analysis* (TGA) combines [Inp.] change information from the version control system with test coverage from automated and manual tests. [Benef.] It highlights which methods (or equivalents in other prog. languages) have been changed since a certain baseline but were not tested afterwards. The analysis results are usually depicted in a treemap (Fig. 1).

**Industrial Experience.** One benefit investigation of TGA was recently published in 2020 [8]. The publication evaluates benefits of TGA usage on a large enterprise application portfolio. As baseline serves the study of Eder et al. [2] with systems from the same portfolio as study objects. Before TGA has been introduced to the projects, around half of the changed methods were released without an execution during the tests. Today, due to TGA, the percentage of untested changes amounts to 10% or less. The application of TGA has caused a decrease in the percentage of field bugs in changed code from 60% to 28%, so by more than half.

### 2.1 System Test Gaps

Scoping Test Gap analysis to the *whole system* [Purp.] allows a view that shows for all changes in the codebase within a certain time frame which additions or modifications have not been run in the tests. [Benef.] It enables a fact-based decision on whether the risk of current gaps in testing are acceptable. For example, shortly before publishing a release $x$ as successor of release $x-1$, TGA makes untested code changes since the last release transparent which helps to *decide* whether the release can be published, or additional tests need to be performed to lower the risk of field bugs.

**Visualization.** A major reason why TGA is attractive for its users lies in its visualization. This makes the results of the analysis tangible and allows to drill down to conspicuous parts of the program. A common view of the results are treemaps. Methods are represented in the diagram as rectangles, with the size of the rectangle representing the length (SLOC) of a method. The color of a rectangle shows whether the method has been changed (*colored*) or not (*grey*) since the baseline. The color code also distinguishes whether a method has been newly added (*red*) or an existing one has been modified (*yellow*). Methods in *green* are either new or changed and have been executed in a test run after the latest change. Teamscale's implementation of this view is interactive. Users can move the cursor over individual rectangles and a tooltip displays metadata and the TGA result of the method. Source code that resides in the same folder is displayed side by side in the diagram, thus the treemap provides a quick overview of whether there are particularly noticeable Test Gaps in certain areas of the system. [Targ.] This view is particularly useful for persons that are responsible for the overall testing of an application, for example, test managers.

**Industrial Experience.** Figure 1 shows the Test Gap treemap of a large business information system shortly before an upcoming release. TGA uncovered a large amount of new code being untested. Investigation revealed that this was induced by miscommunication between different teams. It was decided to shift the release date and make up for the missing tests. After a successful re-testing phase, the application was released having the most important gaps closed.

### 2.2 Issue Test Gaps

TGA can also be applied using a more focused scope, both in terms of time and changes/system parts considered. *Issue Test Gap analysis* combines [Inp.] information from the version control system and from testing with data from an issue tracker. [Purp.] This makes sense in many modern development processes in which responsibility is divided among sub-areas of a system. Rather than (re)viewing *all* Test Gaps of a system, in everyday life, the question is usually »Have I tested the feature, bug fix or other unit of work that I am supposed to verify in sufficient detail?«, asked by both developers and testers. The former, who are responsible for writing automated unit tests need to decide whether their changes are sufficiently tested and ready to be merged into the main branch. The latter, colleagues who run manual tests, follow a test description and can without TGA not be sure that undocumented changes will slip through the test process. [Benef.] Issue Test Gap analysis allows both a quick overview over a set of issues (*Sect. 2.2.1*) and to investigate the Test Gaps of a single ticket in detail (*Sect. 2.2.2*).

#### 2.2.1 Overview Test Gaps of Many Issues

**Visualization.** Approaching Issue Test Gap from a numerical perspective, it reveals the proportion of untested methods that have been *changed* (incl. *added*) during the development of an issue. These proportions can be displayed as horizontal bar charts (see Fig. 2,

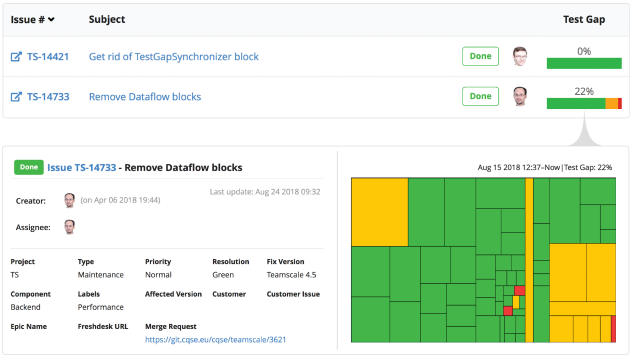


Figure 3: A Test Gap treemap can be generated for a single ticket (lower right side). The color coding is the same as in Fig. 1. Only modified methods are shown in the *Issue Test Gap treemap*, so, it is smaller and, in agile development, more actionable.

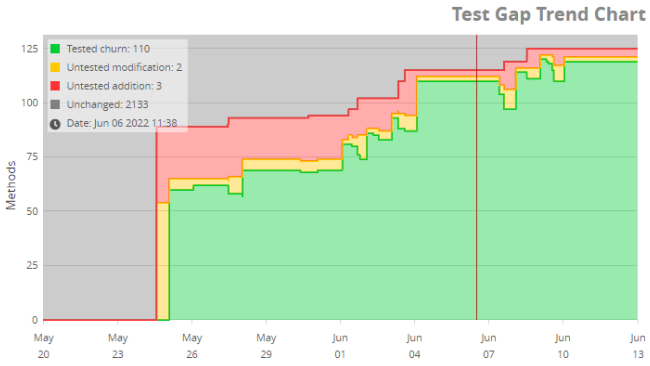


Figure 4: The evolution of Test Gaps over time is shown as a stacked line chart (*Test Gap Trend*). This sample chart shows that on May 24th, methods were changed and added to the system. Shortly thereafter, parts of them were run in the tests. However, some methods remained uncovered and formed a Test Gap. This was reduced over time, so by June 11th there was almost none left.

right side), [Purp.] which allow a quick overview even of many tickets. [Benef.] Larger Test Gaps are immediately visible. In a sensible combination with filters, all tickets of the current iteration can be viewed or all tickets that one has developed themself. A filter could also exclude issues that are still being processed.

**Industrial Experience.** Employees of CQSE, who are in charge of quality engineering, use the view as presented in Figure 2 for quality reports such as *monthly assessments*. Those are brief reports evaluating the code changes and testing efforts of the last month. The quality engineers use the overview in conjunction with *issue queries* (»What are issues that themselves or their belonging parent issues were closed in the past month?«) to identify relevant tickets for which the Test Gap is high.

This shape of TGA and a possible depiction was first published by Rott et al. as *Ticket Coverage* and is now usually referred to as *Issue Test Gap* [13]. In their paper the approach was evaluated on manual executed test cases and the responsible developers were surveyed whether identified Issue Test Gaps would have been needed to be classified as relevant, what they affirmed.

### 2.2.2 Issue Test Gap Treemap

**Visualization.** Changing the point of view to Issue Test Gaps from a quantitative to a qualitative angle, the treemap presentation introduced in Section 2.1 can again be used. [Purp.] All methods included in an Issue Test Gap treemap (Fig. 3) have been changed as part of a ticket; no unchanged methods are shown. Again, only green methods have been executed in the tests after their latest modification. The charm of this visualization is that it is reduced to the essentials [Targ.] for someone who is responsible for testing a single issue. [Benef.] If Test Gaps remain after writing automated tests or after executing a corresponding test plan, these can be quickly recognized as familiar and not as unknown program parts (as many gaps would be in a system-wide TGA). Developers who write automated tests for their application code, know the methods that can appear as Test Gaps and can extend their tests fast. Manual testers can form a feedback loop with the developers until all gaps are closed. The treemap is best integrated into a view that carries the issue context and shows other relevant data as *assignee*, *issue description*, *comments*, *associated commits*, . . . to gather all information in a single location.

## 2.3 Test Gap Trend

The ratio of untested code changes should not increase. Thus it is important [Purp.] to monitor the evolution of Test Gaps in a system over time. A means to do so is the *Test Gap Trend* chart (Fig. 4). [Benef.] On the one hand, this answers whether set goals for improvement are met or whether they are being approached. On the other hand, deteriorations in the coverage, which can lead to a larger Test Gap can be detected quickly.

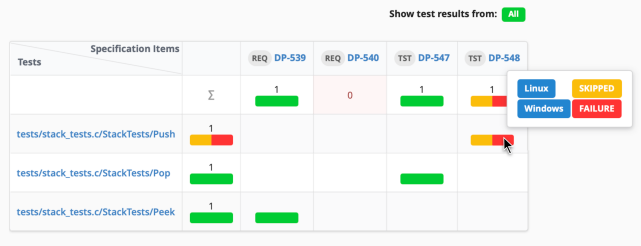


Figure 5: This *verification matrix* was automatically generated by Teamscale. Linkage of test cases *(rows)* to requirements *(columns)* and integration of test data into the tool enabled this. It reveals that requirement *DP-540* is not checked by any test and that the test for *DP-548* is skipped on one test environment and fails on the other.

## 3 Test Results in Verification Matrix

A verification matrix provides an overview over which requirements are covered by which tests and, for example, whether tests succeed on every platform. [Purp.] It answers further questions like:

- »Are there any requirements without associated tests?«
- »Which tests verify too many requirements?«
- »Which tests of a given requirement fail?«

They are used as part of requirements tracing, which is itself part of requirements management. This, in turn, is heavily used in the engineering of safety-critical systems and prescribed in standards for automotive engineering, development of avionic systems and medical technology applications. [12]

[Inp.] If linkage of requirements, code and tests is well maintained, [Benef.] the verification matrix—which is otherwise laborious to create—can nowadays be generated automatically. Teamscale visualizes the result of this analysis in its web-UI (see Fig. 5) and makes the results further examinable. All test results in the matrix are linked to so-called *test details views* which show a variety of information: *test name*, data from the most recent executions like *duration*, *outcome* (passed/failed + recording from standard error) and which *methods have been executed*.

**Visualization.** An example verification matrix is given in Figure 5. The columns list different requirements and the rows indicate implemented or specified test cases. The filled fields inside the ma-

trix show the test result of a row's test that covers the corresponding requirement in the column. The illustration reveals that requirement *DP-540* is not checked by any test and that the test for *DP-548* is skipped on one test environment and fails on the other.

## 4 Test Case Selection and Prioritization

The selection of test cases and their prioritization are means to still test efficiently, despite ever growing and complex systems. For extensive applications, the test suites are often huge. Countless tests verify that the system behaves as expected. The tests are executed automated or by hand, some can run in parallel and some involve hardware that is of limited availability. No matter how the tests are carried out, in more and more projects the problem arises that the tests take too long to provide feedback in a meaningful time. Fast feedback is especially important for developers that have to fix *their* code. Since the mental model for a code region fades with the passing time, fast feedback makes fixing a recently introduced bug easier for the developer (no anew familiarization). [Purp.] The points above justify the increasing number of different algorithms that have been proposed in the literature to select (test case selection, *TCS*) the most useful tests from a large test suite [9]. [Benef.] With *TCS*, tests that are unlikely to find any failures are not executed, thus time is saved. In addition, a set selected this way can be placed in a specific execution order (test case prioritization, *TCP*) in order to optimize against a target; for example, to cover many different areas of the codebase as quickly as possible. In research, also regarding prioritization, a number of different strategies have been proposed [10]. A combined approach of TCS and TCP is implemented in Teamscale and named *test impact analysis* (TIA).

### 4.1 Which Test Would Cover My Test Gap

**Visualization.** The Issue Test Gap treemap was introduced in Section 2.2.2 and Figure 3. [Inp.] If coverage from past test runs is available per test case, [Benef.] it is possible to »predict« before a subsequent test execution which test is likely to execute all or some of the changed methods. With the extension »Which Test Would Cover My Test Gap?«, in a treemap, methods that have not been tested so far but could be executed by running a »known« test are colored in *mint green* (Fig. 6). Hovering over such a method reveals the names of test cases that covered this method in past runs. If the system under test is solely checked automatically, this form of representation might bring little added value. However, if tests are either also or exclusively carried out manually and if there is uncertainty as to which test will close the Test Gap that is still open, this analysis and visualization can be of great advantage.

Even though ten years ago the importance of finding an expedient way to select manual test cases has been highlighted [7], no solution has emerged as a standard so far and nowadays the question regarding a general approach is still open [3]. We are confident that the presented approach can reveal test cases that would have not been respected without the appropriate tool support.

### 4.2 Showing the Benefits of Test Impact Analysis

TIA, as a combination of TCS and TCP, in general does not require a graphical representation. It works integrated in the build system and calculates [Benef.] a sorted test list to be executed by the automated test runner [Inp.] based on code changes, coverage from previous test runs and a prioritization strategy. [Purp.] Conversely, a visualization exists that highlights how propagation of coverage in a system evolves during the test execution. In case the test execution does not follow any particular strategy, the spread of coverage over different system areas is in most cases inefficient. The visualization to demonstrate this problem is constructed by using animated treemaps and has been used in both scientific and product presentations (example video: *https://cqse.eu/vid/no-tia-testrun*).

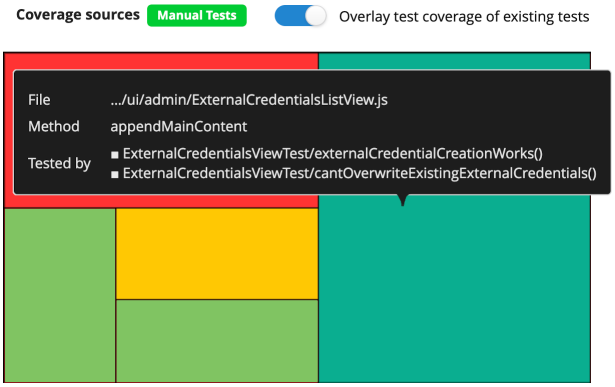


Figure 6: If coverage is available per test case, one can view which *methods could additionally be covered by executing one of the impacted tests*. The four rectangles on the left side follow the known color code in Test Gap treemaps. Without enabling the option »Overlay test coverage of existing tests« the right rectangle would be displayed as a Test Gap (colored in yellow). In the screenshot the option is enabled and since Teamscale »knows« two tests that have executed the corresponding method in a past run, the method is colored in *mint green* indicating that the Test Gap could–likely–be closed by running one of the tests shown in the tooltip. Especially but not exclusively, this can improve a *manual* software testing process and gives testers valuable information.

**Visualization.** Starting with a treemap that shows the changes that should be tested (Fig. 7 at the top), the coverage of a single test case per frame is added. It becomes clear that many of the tests do not run using the changed methods and that these tests can barely reveal any faults there. In Figure 7 only the test cases with a blue border line cover changes.

For the presented approach and visualization here, we refer to its publication in [6]. Amongst others, this paper reports on the evaluation of TIA on 12 open-source systems and describes the approach to be able to use, on average, only 2% of the execution time of the total test suite to reach the first failing test in a build. They use *time to first failure* as metric since they aim for as short feedback times to developers as possible. With a first test failure in the build, it is clear that a developer has to work on their code again.

### 4.3 Pareto Testing

Pareto Testing takes its name from the Pareto principle. The ulterior motive here is that the 80/20 rule also applies to testing—that is, [Purp.] by using a fixed fraction of the total execution time of the test suite, a large part of the overall possible coverage is gained. In contrast to TIA, favorable tests are not always selected on the basis of current changes, but rather [Benef.] a longer-valid test execution list is *calculated once* [Inp.] based on the coverage of past test executions. Pareto Testing is therefore primarily carried out when updating the test list is not feasible for each test run (e.g., because of technical reasons).

**Visualization.** The Pareto Testing chart (Fig. 8) shows how much coverage can be achieved using a specified time budget, in relation to the execution of the entire test suite. The slider is used to set how much testing time is available. As a result [Targ.] a test manager can see which tests should be run and which relative method coverage of the system can be reached.

**Industrial Experience.** An experience report has been published in the article [11]. Before Pareto Testing was introduced, the nightly execution of integration tests took approximately 12 hours. With Pareto Testing, this problem was tackled and a reduced and sorted test list was generated. With a runtime of approximately one hour, this covers 99.2% of all methods the entire suite would cover. Instead

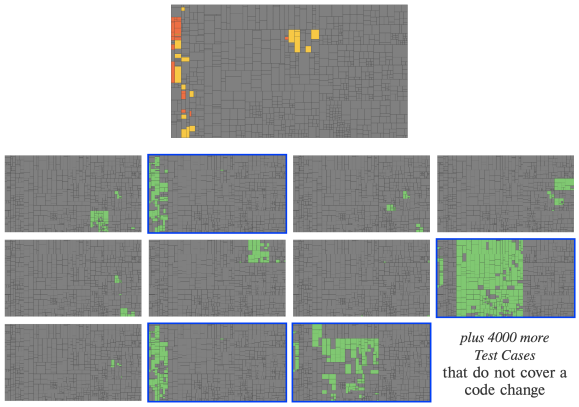


Figure 7: This collage briefly visualizes the concept of the test selection mechanism in Teamscale's *test impact analysis*. Given the set of changed methods (upper treemap), it is checked which of the previously ran tests (the other 11 treemaps) executed some of the changed methods. Only tests that covered the changed methods in the past (blue border) are selected to be run.

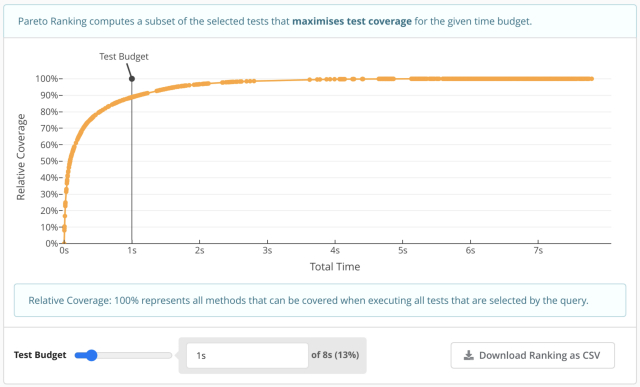


Figure 8: This graph visualizes the application of *Pareto Testing*: How much method coverage compared to the whole test suite can be achieved with a limited testing time budget. According to the available time (can be set using the slider) and coverage from previous test runs, an optimized test execution list is generated. As far as possible, this approach maximizes a broad coverage over different code areas of the software system.

of having to wait until the following day, developers now receive feedback on their work much faster. In addition, the entire test suite is run overnight to ensure that no test failures go unnoticed.

## 5 Find Code that Caused a Test Failure

When a test case fails, it is beneficial if one can identify methods covered by the test that have been modified since the last successful run. This is because the reason of the failure is likely within the recent changes and thus [Benef.] the starting point of where to improve the code and fix the test. [Inp.] Prerequisite is the recording of coverage per test case, which can be combined with the change information since the last test pass. [Purp.] When test suites take a long time to execute, they often run over the weekend. After the test suite has finished, (say Monday) someone will see that a number of test cases failed. However, since the test suite did not run over a specific changeset, but over all changes from the previous week, it is difficult to find out which change actually caused the test failure.

**Visualization.** Figure 9 shows the result presentation of this analysis in Teamscale. The treemap shows methods that have either

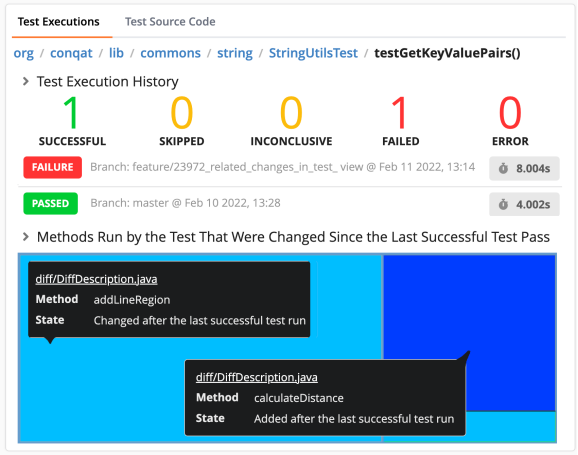


Figure 9: In Teamscale's »Test Details View«, a treemap shows the methods that have been *changed* (light blue) or *added* (blue) *since the last successful test run*. It can be used to identify the reason of a test failure as the cause is likely within the recent changes. In case of lacking this information a manual inspection has to be performed which is laborious especially when the test suite is large and dependencies between tests and code changes are not clear.

been changed (light blue) or added (blue) since the last successful run of the test case. This chart is embedded into a view that summarizes data related to a single test case, such as a *test execution history* (results of the recent test runs), *coverage of the test run* and (if the testing framework is supported) the *source code* of the test.

## 6 Charts Revealing Interrelations

[Purp.] »What are the implications of certain areas of the codebase being less tested than others? Can one spot differences?« These questions do not only arise when it comes to auditing systems, root-cause analyses or to convincing people of certain analyses. In day-to-day development, too, it is important to keep certain connections in mind. [Targ.] Researchers can profit from interrelations analyses just like managers and any other project stakeholder.

Two analyses and their forms of representation are given in the following. With their key takeaways they have a strong educational effect. [Inp.] Both times, test coverage is compared to other metrics of the system. First, the frequency with which a code area needs to be touched in a bug ticket, and second, the overall frequency of changes—at different code- and coverage granularities, and with different visualizations.

### 6.1 Coverage and the Frequency of Being Subject in a Bug Ticket

**Visualization.** The comparison of two treemaps can be used to check the interrelations of different metric values of certain areas in the source code of a system. For this purpose, the rectangles in both treemaps can be colored with different degrees of opacity according to the measured value. Is the same area in both treemaps noticeable—for example very dark in one and light in the other treemap, this argues for a relation.

This can be used for various investigations. [Inp.] Frequently, the change rate of files or sources belonging to a particular module is of interest. [Purp.] When focusing on changes that are made as part of issues which are classified as *bug tickets*, this count represents a measure of *defect density*. If the defect density is particularly high somewhere, a lot of expenditure will probably flow into this area, and it is worth taking countermeasures. But against what? It might be that many faults slip through simply because the coverage is low,

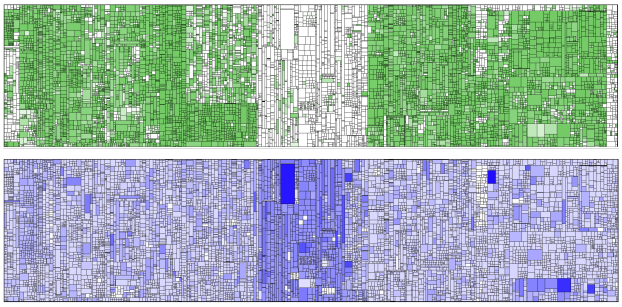

Figure 10: In these two treemaps, rectangles represent source code files. The shade of color in the upper treemap is darker, the higher the test coverage of the respective file is. The lower treemap shows for the same system how often certain files were changed in the context of *bug tickets*; the darker the blue is, the higher the number of changes. The center area of the treemaps is striking and reveals clearly an interrelation: The area of the system that shows the lowest code coverage needed the most bugfixes.

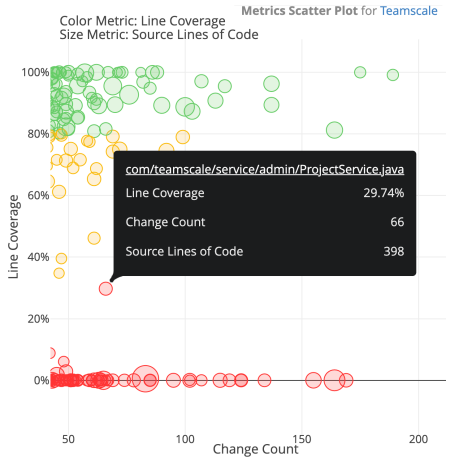


Figure 11: A scatter plot can set multiple measures of a system in relation. Beside the location (x- and y-coordinates) of a circle depicting an entity also the size and the color of the circle can encode information. In this case, line coverage within a certain file is set in relation to how often the file is changed, additionally the circle size depicts the file size (SLOC). The plot allows to detect files that are often changed and have at the same time a low line coverage. Due to its interactivity in the UI, a drill down to striking files is easily possible.

that is, more and better tests are needed. Just another possible reason is poor code quality which can be further investigated and controlled using static code analyses. [Benef.] If one has identified a reason for the high defect density, a decision can be made for further work on it. In the case of a bad code structure, for example, refactorings can help to make the code more maintainable.

**Industrial experience.** In the treemaps shown in Figure 10, the center area of the treemaps is striking. The upper diagram shows line coverage in percent. The layout of the diagram below is in conformity with the upper one, i.e., same files are drawn in the same location. There, however, the coloring shows the frequency of how often a file had to be changed as part of a bug ticket. The darker area below lines up with the almost white area above—it was the UI code of the analyzed system. A survey among the development team revealed that testing the UI was often avoided because of a bad testability at that time; UI tests were difficult to write. Only with the help of these two treemaps which made the problem very apparent, it was addressed. The decision was taken to refactor the UI code to make it more testable.

### 6.2 Coverage and Measures from Static Analyses

[Purp.] Context-dependent, a comparison of a system's metric values in a scatter plot can be more forthright. [Benef.] An interactive map that puts measures of the system in relation can help, especially when it comes to uncovering *outlier classes or files* in a system.

**Visualization.** We stay with the example from the last section, [Inp.] coverage and change frequency. In the scatter plot in Figure 11, each file is represented as a circle. The larger the circle, the longer (SLOC) the corresponding file. Line coverage is plotted on the y-axis, the frequency of change on the x-axis. The color of the circles also displays the line coverage of the files separated by thresholds—files with high coverage in green, medium coverage in yellow and low coverage in red. In general, the color semantics, as well as the metrics that are plotted on the two axes, can be freely configured. The color could, for example, also indicate how many different committers have changed a file or how high the finding density from static code analyses is.

## 7 Conclusion and Future Work

We presented various analyses and their visualizations. Each one of them allows the existing test processes in software development to be scrutinized and improved. In practice, we see the need for this, because systems are becoming more and more extensive nowadays and it is not always possible to achieve improvements in testing simply through more tests, more parallelization or more hardware, but the actual causes of the problems must be combated. From industry cooperation, we know that projects which use one or more of these analyses report positive on the benefits. The advantage seems to be especially high if *(1)* teams receive support from an experienced quality engineer and *(2)* establish a continuous process to regularly check on their code and test quality.

For the future, we would like the use of the analyses to be embedded simpler in the development process, so that even more developers, testers, etc. can benefit from them, and ultimately the software quality of systems they work with can be pushed to a higher level. In practice, we have found that, today, many persons in charge are unfamiliar with analyses such as we presented in this paper, yet. Therefore, artifacts from software development and testing are not adequately used to achieve improvements in future engineering.

From a scientific point of view, further studies are desirable in addition to the existing ones in order to further prove the effects of using the analyses in diverse development environments. In addition the capabilities of other tools should be examined and how they compare to the here presented ones implemented in Teamscale.

It should be explicitly emphasized that the advantages are not limited to a few sub-sectors of the economy for which software is being developed, but that effects can be found equally in, for example, the development of hospital software, mobile app development and embedded software development.


## Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "Q-Soft, 01IS22001A". The responsibility for this article lies with the authors.

Many thanks to CQSE GmbH that provided many of the graphics used in the paper, and to the colleagues who were involved in implementing and generating them based on different audited software systems. This includes but is not limited to: Florian Deißenböck, Florian Dreier, Martin Feilkas, Benjamin Hummel, Elmar Jürgens, Raphael Nömmer, Dennis Pagano and Fabian Streitel.

We also thank Roman Haas for his valuable review and Andreas Göb for the impulse to write this paper.

## REFERENCES


[1] S. Amann and E. Jürgens. Change-Driven Testing. In S. Goericke, editor, *The Future of Software Quality Assurance*, chapter 1. Springer Nature, 2019.

[2] S. Eder, B. Hauptmann, M. Junker, E. Jürgens, R. Vaas, and K.-H. Prommer. Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, 2013.

[3] R. Haas, D. Elsner, E. Jürgens, A. Pretschner, and S. Apel. How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1281–1291, 2021.

[4] R. Haas, R. Niedermayr, and E. Jürgens. Teamscale: Tackle Technical Debt and Control the Quality of Your Software. In *Proceedings of the 2nd International Conference on Technical Debt (TechDebt'19)*. IEEE, 2019.

[5] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale. Software testing: Survey of the industry practices. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1449–1454. IEEE, 2018.

[6] E. Jürgens, D. Pagano, and A. Göb. Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites. *Whitepaper, CQSE GmbH*, 2018.

[7] E. Jürgens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke. Regression Test Selection of Manual System Tests in Practice. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 309–312. IEEE, 2011.

[8] E. Jürgens, U. Proft, and J. Rott. Benefits of TGA at Munich Re. In *CQSE Spotlight #2/2020*, page 5. CQSE GmbH, 2020. https://cqse.eu/en/spotlight-2020-2.

[9] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani. Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)*, 50(2):1–32, 2017.

[10] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.

[11] R. Nömmer and J. Rott. Pareto Testing at BVK. In *CQSE Spotlight #2/2021*, page 9. CQSE GmbH, 2021. https://cqse.eu/en/spotlight-2021-2.

[12] D. Pagano, M. Feilkas, J. Rott, and D. Transiskus-Riering. Agile Requirements Tracing. In *CQSE Spotlight #1/2021*, pages 6–7. CQSE GmbH, 2021. https://cqse.eu/en/spotlight-2021-1.

[13] J. Rott, R. Niedermayr, E. Jürgens, and D. Pagano. Ticket Coverage: Putting Test Coverage into Context. In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM'17)*, 2017.