# Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice

Elmar Jürgens
CQSE GmbH
juergens@cqse.eu

Dennis Pagano
CQSE GmbH
pagano@cqse.eu

***Abstract*—In long-lived systems, errors typically occur in code areas that have been frequently changed. As a consequence, test managers put great emphasis on thoroughly testing modified code. However, our studies show that even in well-structured testing processes code changes often accidentally remain untested.**

**Test Gap analysis finds untested changes and gives testers the chance to test them in a timely manner. As a consequence, Test Gap analysis allows you to effectively monitor your testing processes.**

**After an introduction to Test Gap analysis we describe our experiences both with the usage of this tool at our customers and during our own development over the last few years. We also go into detail on how Test Gap analysis can be applied in different phases of the testing process.**

## How well are code changes actually covered by tests in practice?

In many systems manual test cases still make up the majority of all tests. In a large system it is anything but trivial to choose manual test cases that actually run through the changes that were made since the last test phase and presumably contain most bugs.

In order to better understand whether the tests actually reached those changes, we conducted a scientific study[1] on an enterprise information system. The examined system comprises approximately 340,000 lines of C# code. We conducted the study over a period of 14 months of development and analyzed two consecutive releases.

Through static analyses we determined which parts of the source code were newly developed or changed for both releases. For both releases about 15% of the source code was modified. Furthermore, we analyzed all testing activities. In order to do so, we tracked the test coverage of all automated and manual tests over the course of several months.

When evaluating the combination of modification and test data, it became apparent that **approximately half the changes went into production untested** – despite a systematically planned and executed testing process.

## What are the consequences of untested changes?

To quantify the consequences of untested changes for users of the system, we retrospectively analyzed all errors that occurred in the months following the releases. This brought to light that changed, untested code is five times more error-prone than unchanged code (and also more error-prone than changed and tested code).

Our study makes plain that in practice changes very frequently reach production untested and that they cause the majority of field bugs. However, we thus observe a concrete starting point to systematically improving test quality: if we manage to test changes more reliably.

## How does changed code escape the test?

The amount of untested production code actually surprised us when we conducted this study for the first time. In the meantime, comparable analyses have been performed in several systems, using several programming languages and in different companies, and the results are often similar. The causes of these untested changes are, however, – to the contrary of what you may assume – not a lack of discipline or commitment on the tester's part but rather the fact that without suitable analyses it is extremely hard to reliably catch changed code when testing large systems.

When selecting cases, test managers often orient themselves on changes that are documented in the issue tracker (Jira, TFS, Redmine, Bugzilla, etc.). In our experience, this works well for changes made for functional reasons. Test cases for manual tests usually describe interaction sequences via the user interface to test specific functionality. If the issue tracker contains changes to a functionality the corresponding functional test cases are selected for execution.

However, we have learned that there are two reasons for issue trackers not being suitable sources of information in consistently finding changes. Firstly, changes are often technically motivated, for example, clean-up operations or adaptations to new versions of libraries or interfaces to external systems. These technical changes do not make it clear to the tester which functional test cases need to be executed to run through the changed code.

Secondly and more importantly, issue trackers often miss essential changes, be it due to time pressure or policy reasons. This leads to gaps in the issue tracker data. In order to consistently find changes without gaps we need reliable information about which changes were tested and which were not.

## What can be done?

Test Gap analysis is an approach that combines static and dynamic analysis techniques to identify changed but untested code. It is comprised of the following steps:

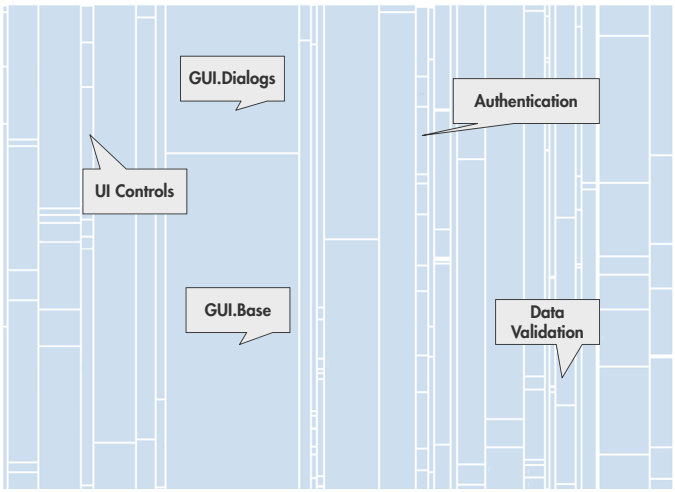


Figure 1. Treemap with the components of the system under test. Each rectangle represents one component. The surface area corresponds to the size of the component in lines of code. As an example, the primary function of individual components is depicted.

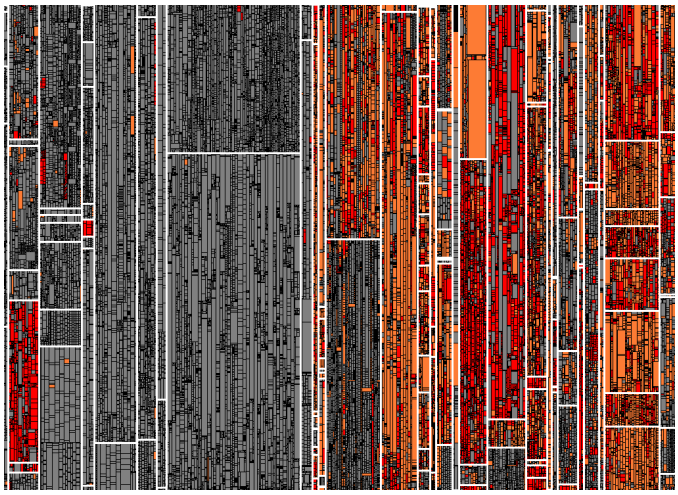

Figure 2. Changes within the system under test since the previous release: Each small rectangle represents one method of the source code. Unchanged methods are depicted in gray, new methods in red and changed methods in orange.

**Static analysis.** A static analysis compares the current state of the source code of the system under test to that of the previous release in order to determine new and changed code areas. In doing so, the analysis is intelligent enough to differentiate between varying kinds of changes. Refactorings, which do not lead to a modification of the behavior of the source code (e.g., changes to documentation, renaming of methods, or code moves), cannot cause errors and can thus be filtered out. This brings our attention to those changes that lead to a change in the behavior of the system.

The changes to one of the systems we analyzed can be seen in Figure 2. Figure 1 serves to illustrate further the parts into which the underlying system is divided.

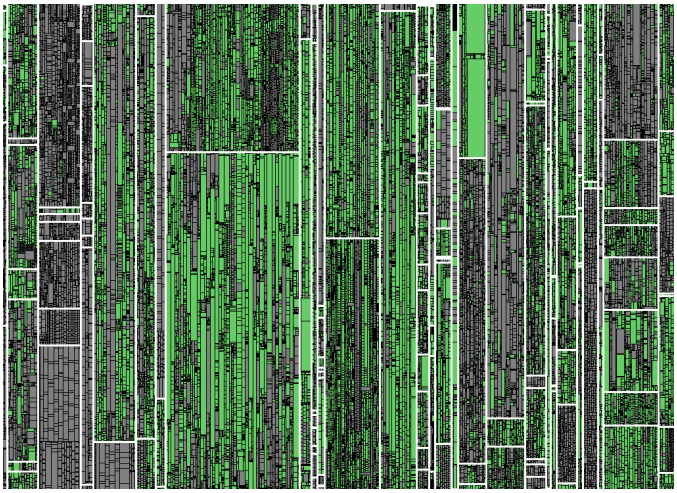

Figure 3. Test coverage in the system under test at the end of the test phase. Untested methods are depicted in gray, tested methods in green.

**Dynamic analysis.** In addition, test coverage is determined with the help of dynamic analyses. The crucial factor here is that all tests are recorded, that is both automated and manually executed test cases. The executed methods are depicted in Figure 3.

**Combination.** Test Gap analysis then detects untested changes by combining the results of the static and dynamic analyses. Figure 4 shows a treemap containing the results of a Test Gap analysis for this system. In this case, the small rectangles in the different components represent the contained methods, their surface area corresponds to the length of the method in lines of code. The colors of the rectangles represent the following:

- Gray methods have remained unchanged since the previous release.
- Green methods were changed (or added) and were executed in any test.
- Orange (and red) methods were changed (or added) and were not executed in any test.

It is plain to see that on the right side of the treemap whole components containing new or changed code were not executed in the testing process. No errors contained in this area can have been found.

## How to use Test Gap analysis

Test Gap analysis is useful when executed continuously, for example, every night, to gain insight each morning into the changes made and tests executed up to the previous evening. For this purpose, dashboards containing information about test gaps are created, as illustrated in Figure 5.

These dashboards allow test managers to decide in a timely manner, whether further test cases are necessary to run through the remaining changes while the test phase is still ongoing. If the effort was successful, this will become evident on the newly calculated dashboards the next day.

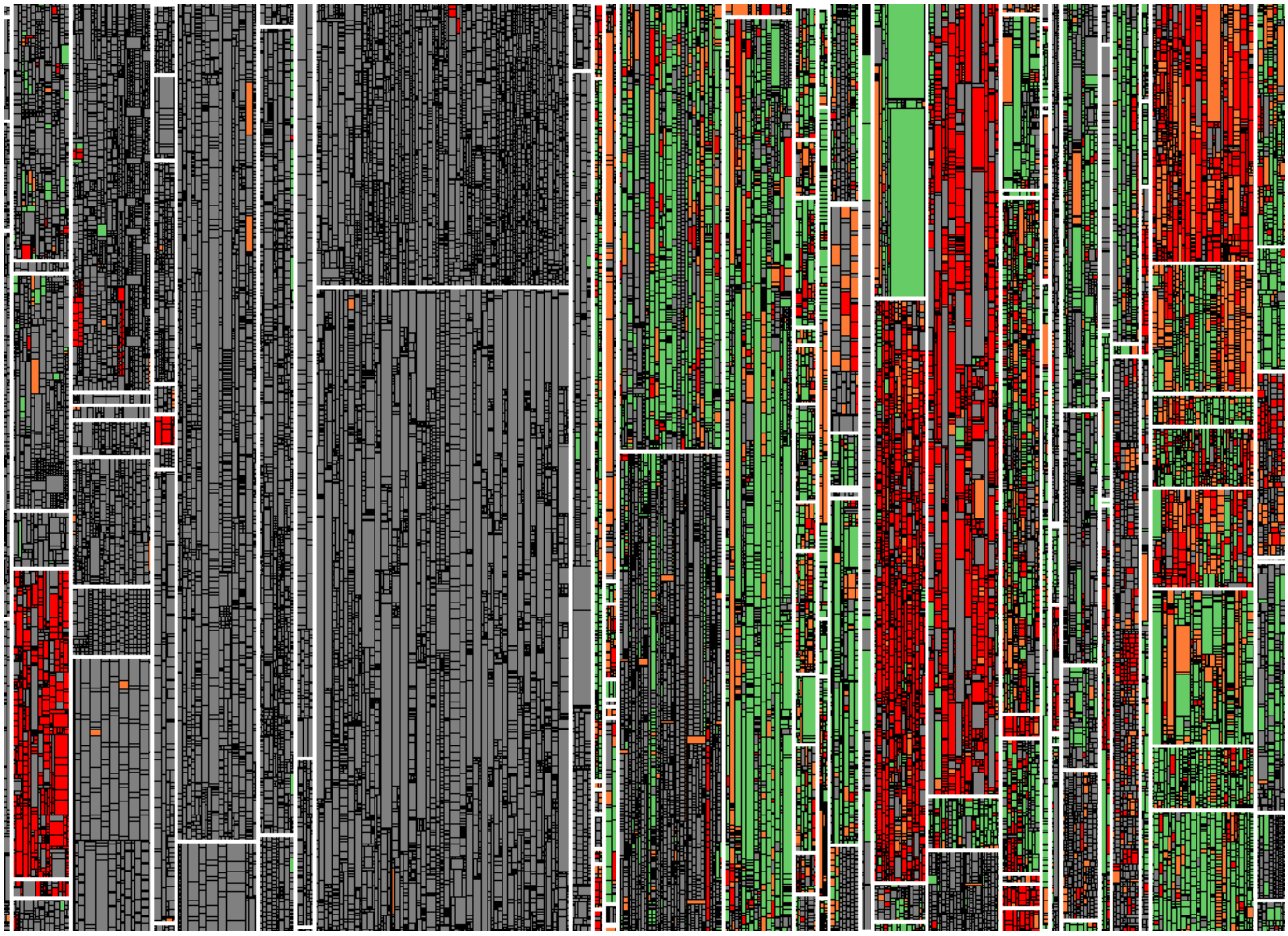

Figure 4. Test gaps at the end of the test phase. Unchanged methods are depicted in gray. Changed methods that were tested are depicted in green. New untested methods are depicted in red, changed untested methods in orange.

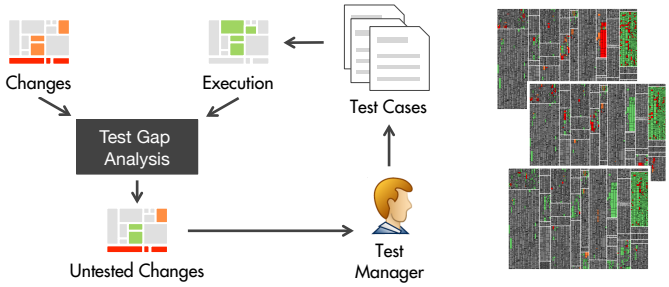


Figure 5. Usage in the testing process.

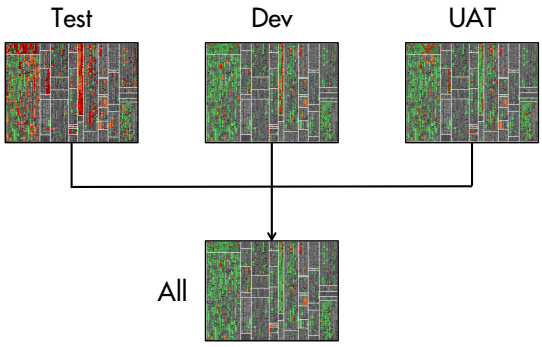


Figure 6. Several dashboards are used for a detailed analysis.

If multiple test environments are in use simultaneously, one dashboard should be created for each individual environment so that test coverage can be specifically assigned to the corresponding dashboard. In addition, there is a dashboard that encompasses all information from all environments. Figure 6 depicts an example with three different test environments:

- **Test.** This is the environment in which testers carry out manual tests.
- **Dev.** In this environment automated test cases are executed.
- **UAT.** The user acceptance test environment is used for end users carrying out exploratory tests with the system under test.
- **All.** Collects the results of all three test environments.

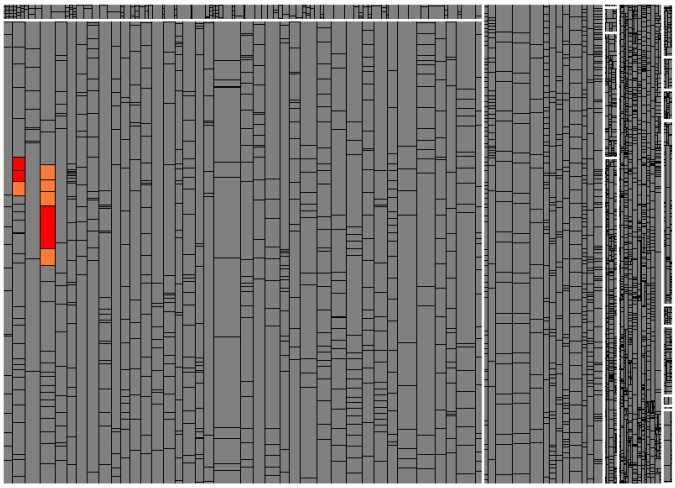

Figure 7. Methods changed during a hotfix.

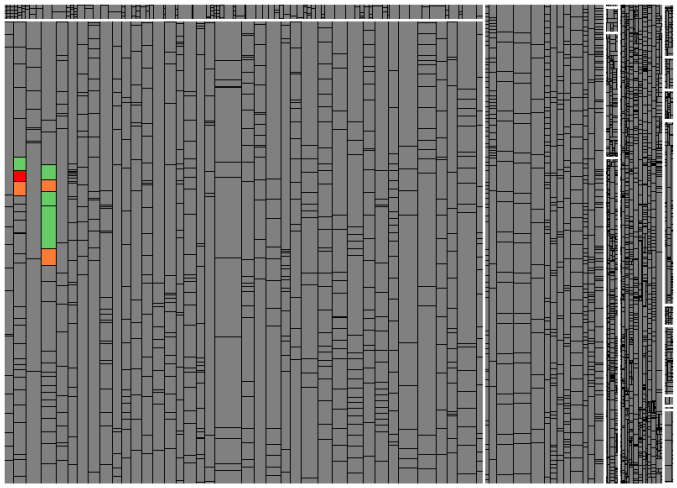

Figure 8. Tested and untested methods during confirmation testing of a hotfix.

### Which projects benefit from the use of Test Gap analysis?

We have used Test Gap analysis on a wide range of different projects: from enterprise information systems to embedded software, from C/C++ to Java, C#, Python and even ABAP. Factors that affect the complexity of the introduction include, among others:

- **Execution environment.** Virtual machines (e.g., Java, C#, ABAP) facilitate the collection of test coverage data.
- **Architecture.** For server-based applications, test coverage data has to be collected on fewer machines than for fat-client applications.
- **Testing process.** Clearly defined test phases facilitate planning and monitoring.

### What are the advantages of Test Gap analysis during hotfix testing?

Usually there is very little time to test hotfixes. The objectives of hotfix tests are to ensure that the fixed error does not re-occur as well as making sure that no new errors have been introduced. In the latter case, there should at least be a guarantee that all changes made in the course of the hotfix were tested. For this purpose, we define the last release as the reference version for Test Gap analysis and detect all changes made due to the hotfix (for example, on its own branch) as shown in Figure 7.

With the help of Test Gap analysis we then determine whether all changes were actually tested during confirmation testing. The example in Figure 8 demonstrates that a part of the methods is still untested. Our experience shows that Test Gap analysis specifically helps to avoid new errors that are introduced through changes during hotfix tests.

### What are the advantages of Test Gap analysis during a release test?

In this article, a release test is defined as the test phase prior to a major release, which usually involves both testing newly implemented functionality and executing regression tests. For this, different kinds of tests are frequently used.

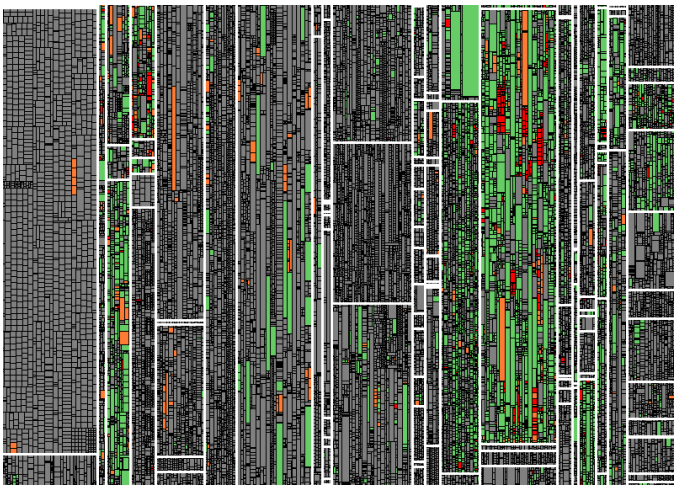

Figure 9. Fewer test gaps when Test Gap analysis is used continuously.

Our experience shows that Test Gap analysis significantly reduces the amount of untested changes that reach production.

Figure 9 depicts a test gap treemap of the same system that can be seen in Figure 4. While Figure 4 was determined retrospectively, Figure 9 is a snapshot of an iteration in which Test Gap analysis is applied continuously. In this snapshot, both manual as well as automated tests are taken into account. It is plain to see that it contains much fewer test gaps.

We have also observed that in many cases some test gaps are accepted, for example, when the corresponding source code is not yet available via the user interface. The key, however, is that these are conscious and well-founded decisions with predictable consequences.

## What are the limitations of Test Gap analysis?

Like any analysis method, Test Gap analysis has its limitations. To know them is crucial in making the best use of the analysis.

One of the limitations of Test Gap analysis are changes that are made on the configuration level without changing the code itself as these changes consequently remain hidden from the analysis.

Another limitation of Test Gap analysis is the significance of covered code. Test Gap analysis evaluates which code was executed during the test. However, the analysis cannot figure out how thoroughly the code was tested. This potentially leads to undetected errors despite the analysis depicting the executed code as "green". This effect increases with the coarseness of the measurement of code coverage.

However, the reverse is true: red and orange code was not executed at all. Thus, no contained errors can have been found.

Our experience in practice shows that the gaps brought to light when using Test Gap analysis are usually so large that substantial insights into weaknesses in the testing process are gained. Concerning these large gaps, the limitations mentioned above are not significant.

## Outlook

Another exciting field for applying the described analysis methods is the production environment. In this case, the recorded executions are no longer executed test cases but system interactions carried out by the end user. This helps to determine which of the features that were added to the previous release are actually used by the end user. In our experience, the results can frequently be surprising.

Figure 10 illustrates the use of an enterprise information system in the style of a Gantt chart [2]. Each line represents one of the application's features. The x-axis shows the measurement period. On weekends as well as during the Christmas season the system is, as we expected, used less than on other days. However, the figure only depicts those features that were actually used. Yet the analysis showed that 28% of the application's features were not used at all, which came as a surprise to all stakeholders involved. In this case, the analysis led to the deletion of a quarter of the application's source code so that in the following releases less testing efforts were necessary or were applied in a more effective way[1].

## Further Information

At **www.testgap.io** we compiled further materials on Test Gap analysis, such as research projects, blog posts and tool support. Furthermore, as the authors of this article, we are always happy to receive e-mails with questions and feedback (also negative) on this text or on Test Gap analysis in general.

[1]For usage analysis on feature level, several methods are necessary that go beyond the ones described in this article. They are described in the referenced paper.

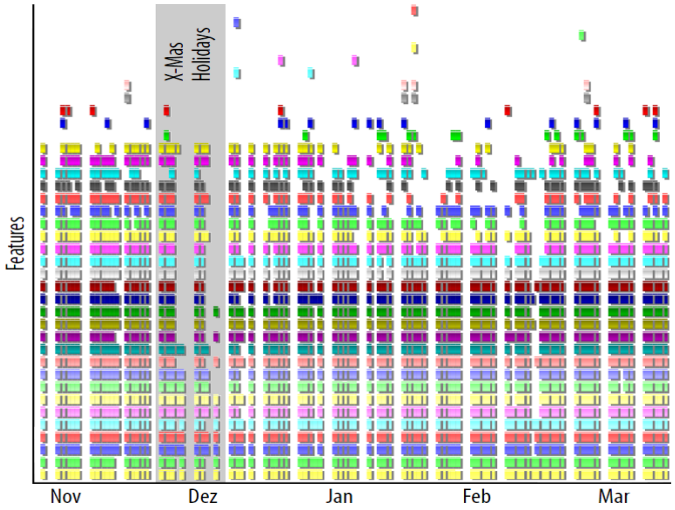


Figure 10. Using the features of an enterprise information system in production.

## References

[1] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Elmar Juergens, Rudolf Vaas, and Karl-Heinz Prommer. 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] Elmar Juergens, Martin Feilkas, Markus Herrmannsdoerfer, Florian Deissenboeck, Rudolf Vaas, and Karl-Heinz Prommer. Feature profiling for evolving systems. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 171–180. IEEE, 2011.