

Erfahrungen aus 10 Jahren

Test-Gap-Analyse im Praxiseinsatz

CQSE

Dr. Sven Amann

Agenda

- Teil 1: Grundlagen der Test-Gap-Analyse
- Teil 2: Herausforderungen bei der Einführung
- Teil 3: Kosten-Nutzen-Berechnung

Teil 1

Grundlagen der Test-Gap-Analyse

Stellen Sie sich vor, Sie sind dafür verantwortlich,
dass alle Codeänderungen »*ausreichend*« getestet werden...

Wo treten Fehler in Produktion auf?

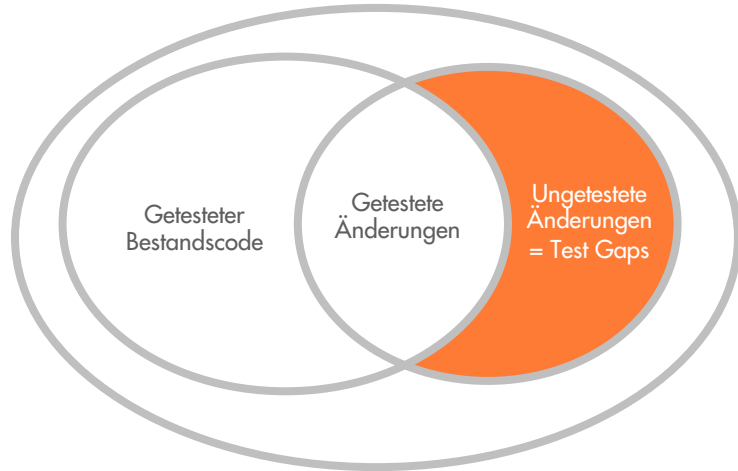
Studie: C# System

Release A:

15% Code neu/geändert,
>50% ungetestet

Release B:

15% Code neu/geändert,
>60% ungetestet

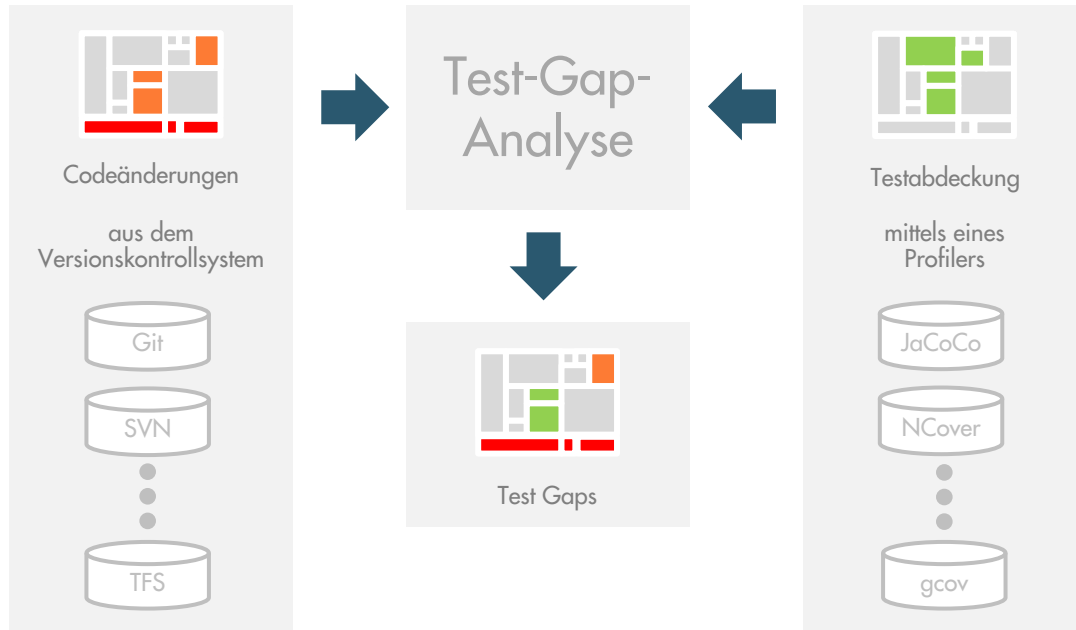


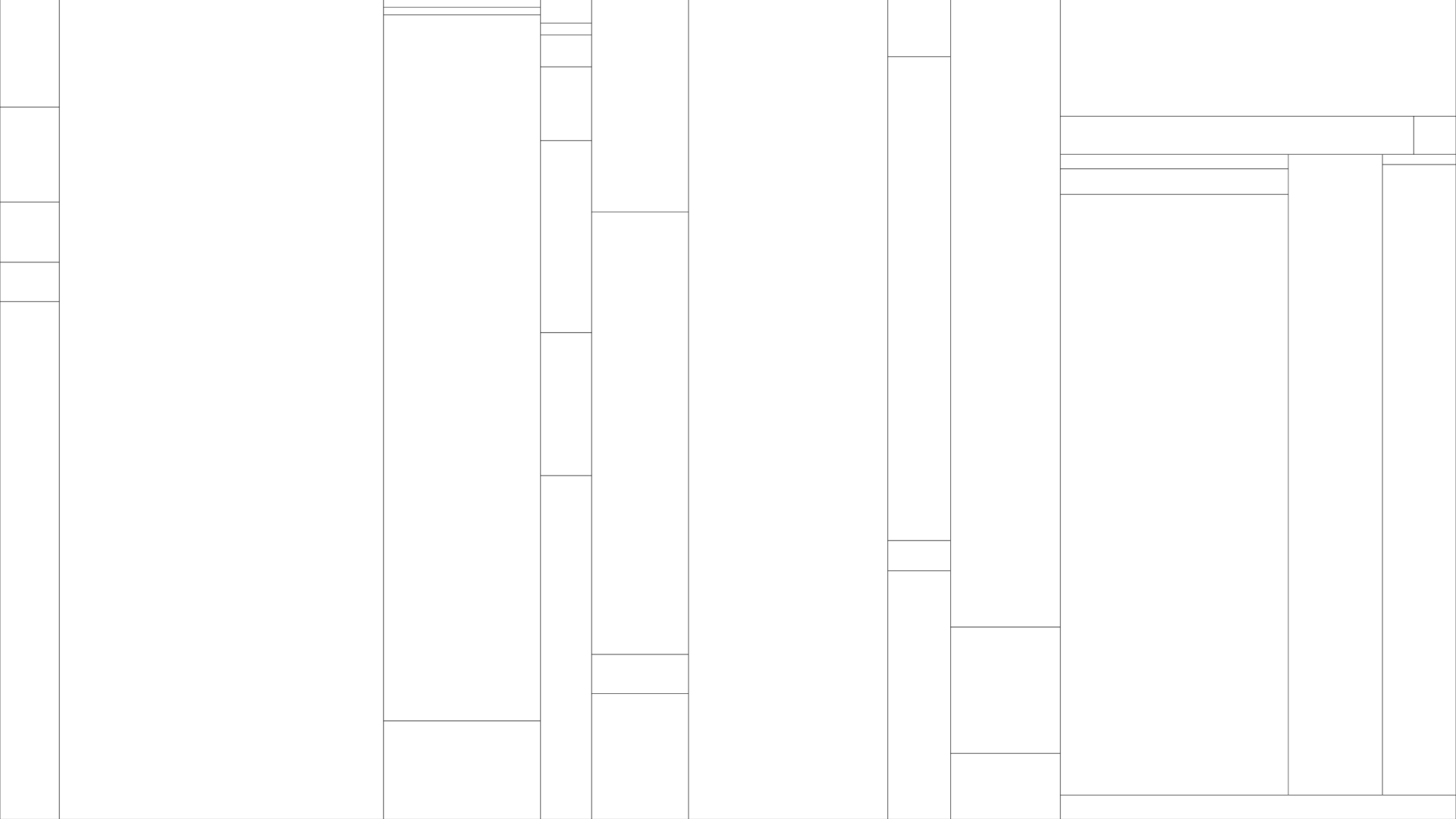
Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!

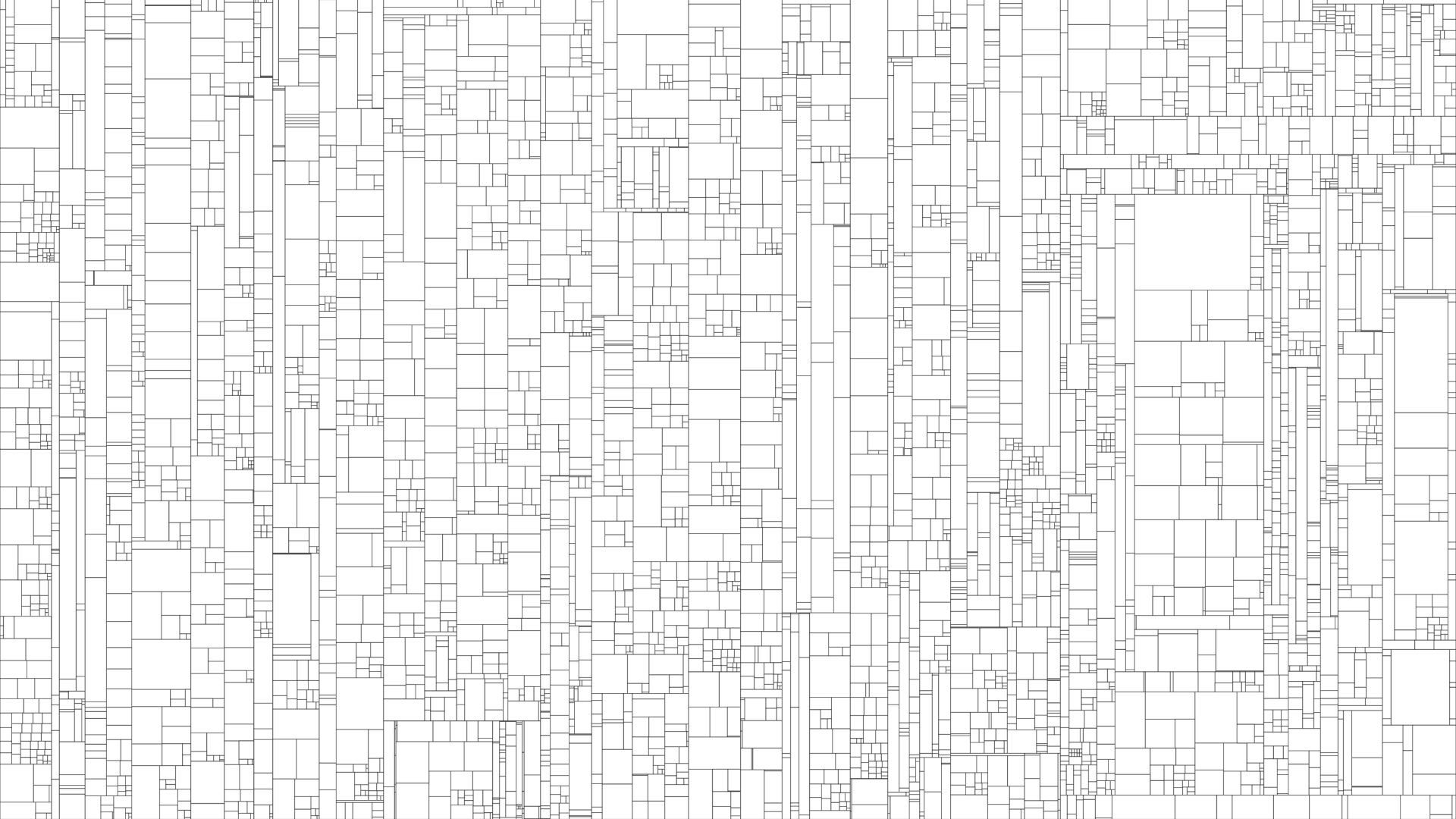
Ziel

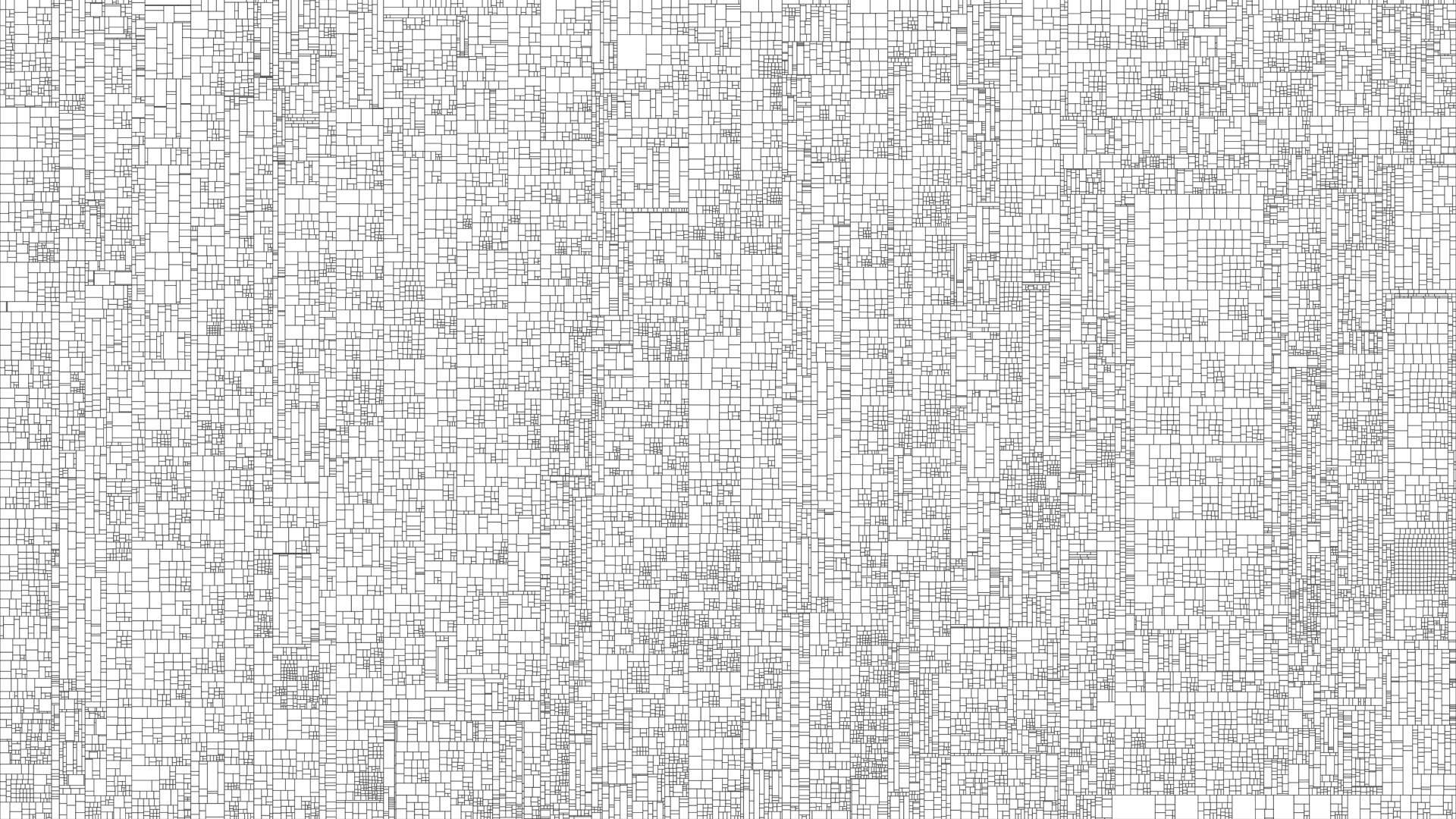
Finde die ungetesteten Änderungen
(= Test Gaps)

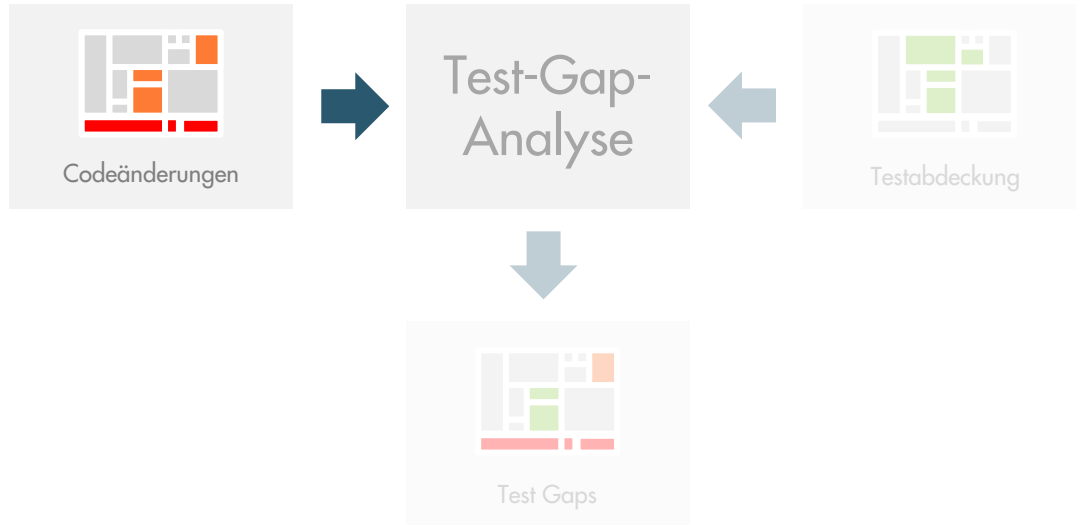
*Weil Fehler im geänderten, ungetesteten Code
sehr viel wahrscheinlicher sind als anderswo*



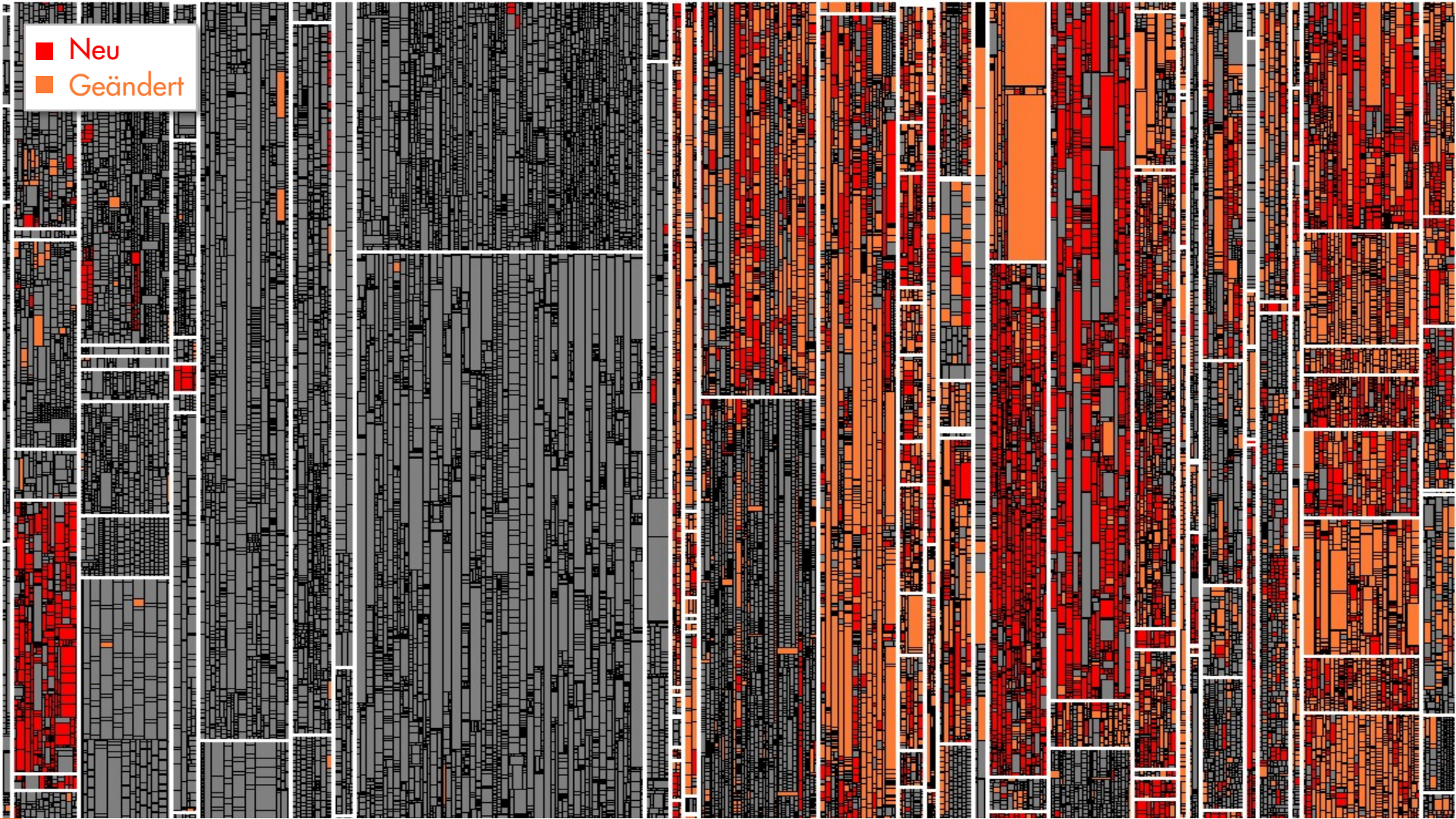


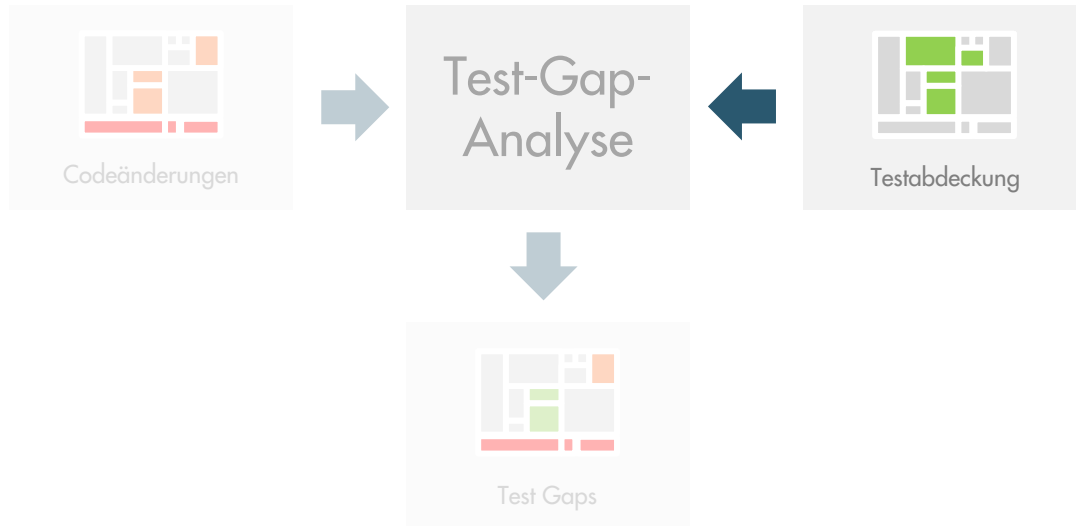






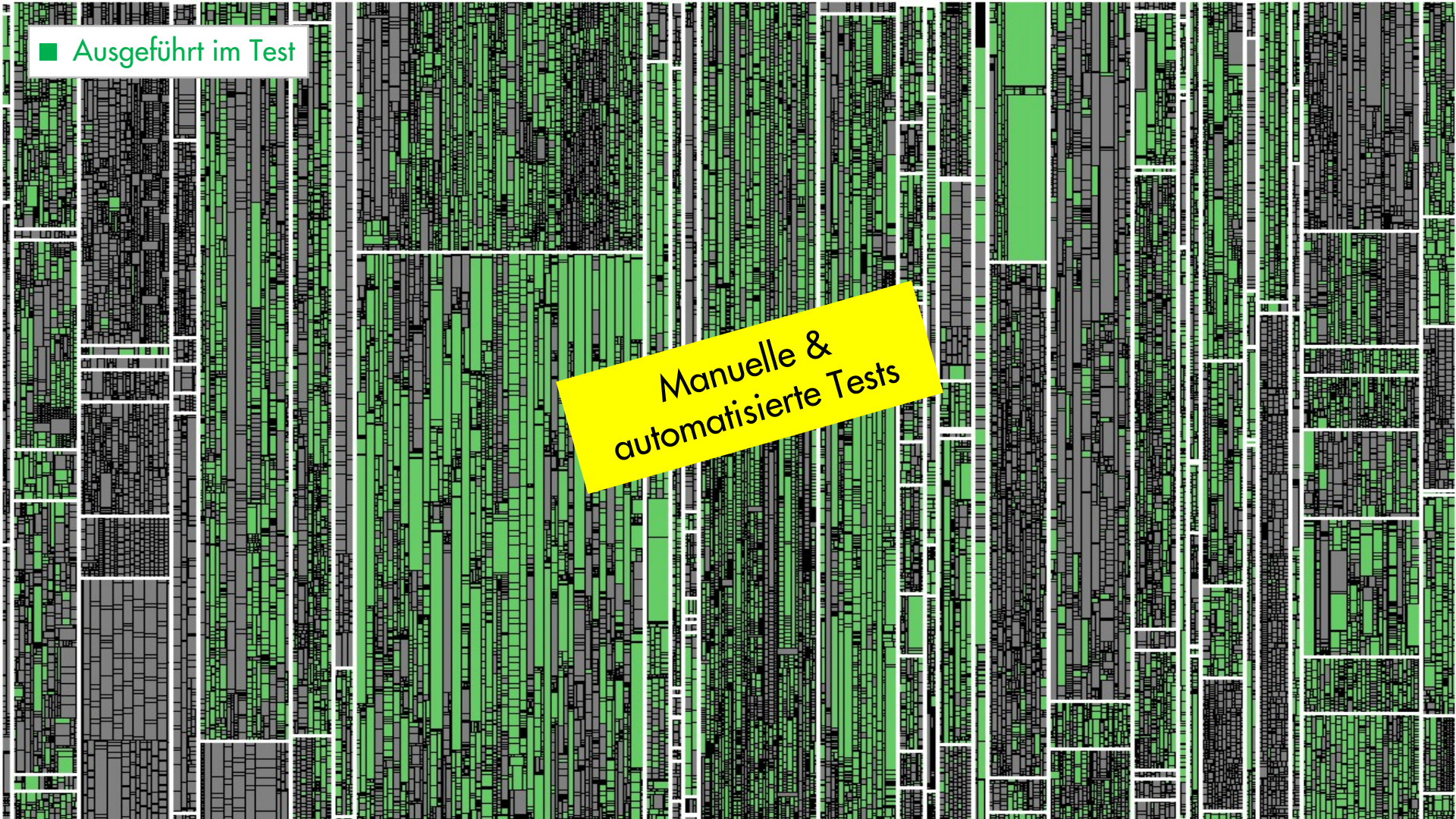
■ Neu
■ Geändert

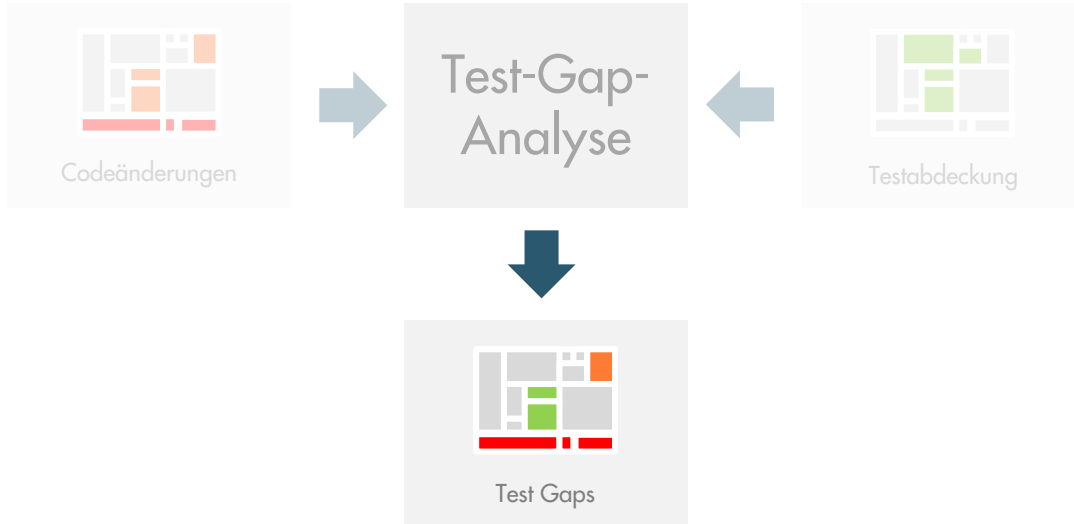




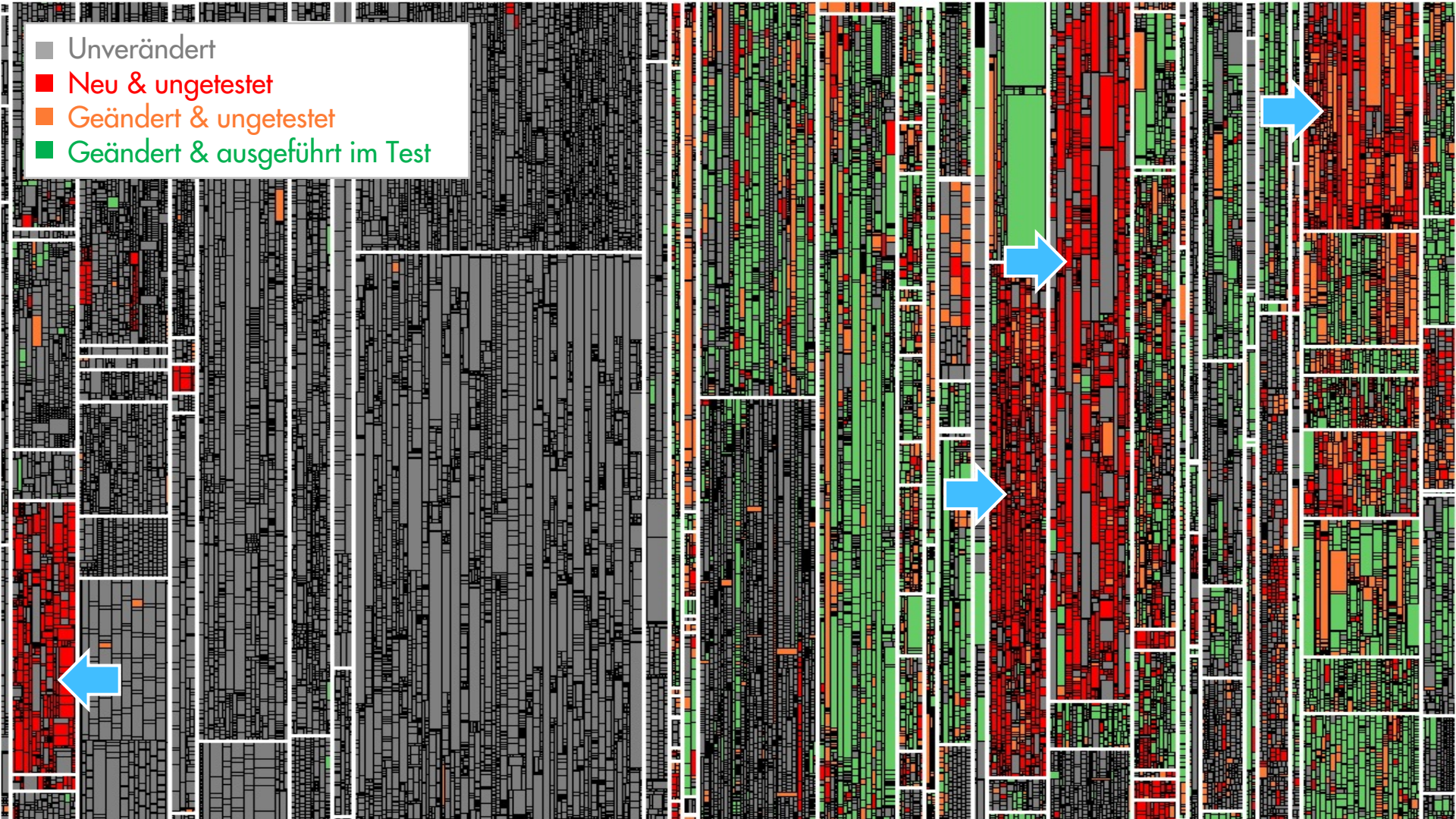
■ Ausgeführt im Test

Manuelle & automatisierte Tests

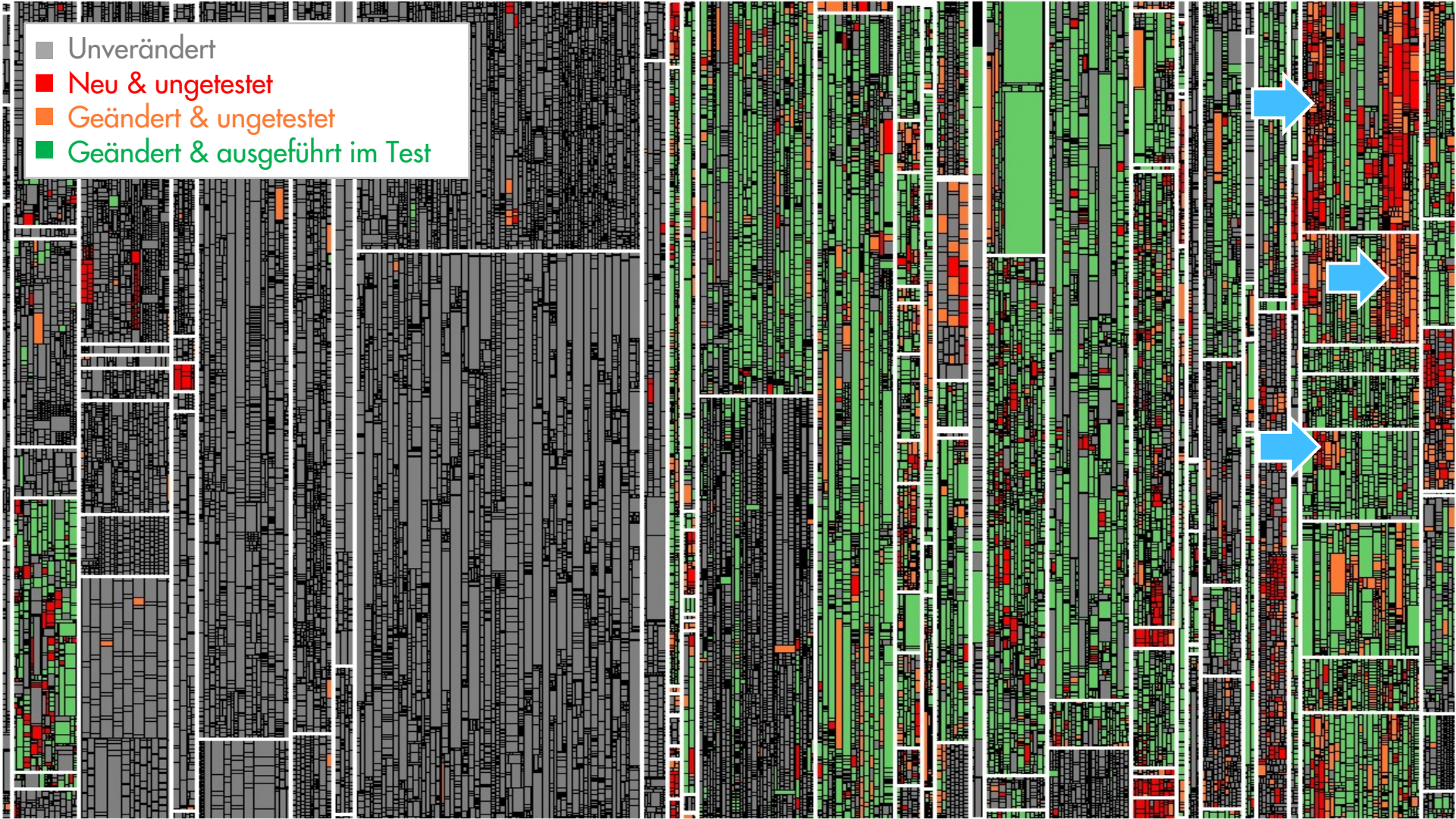




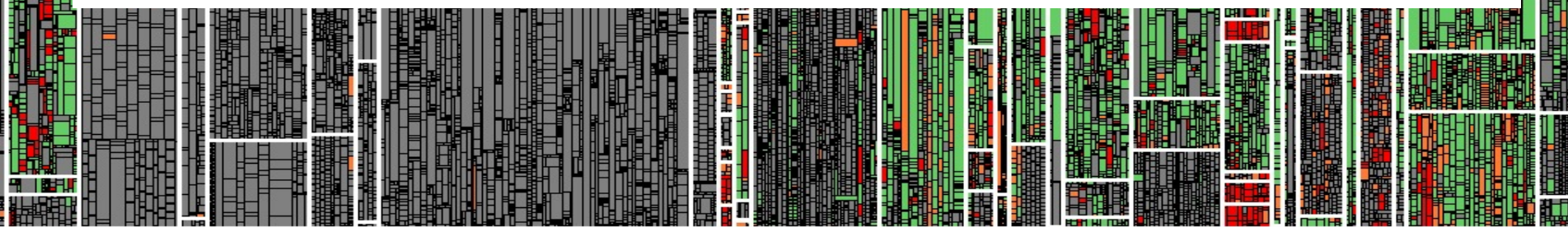
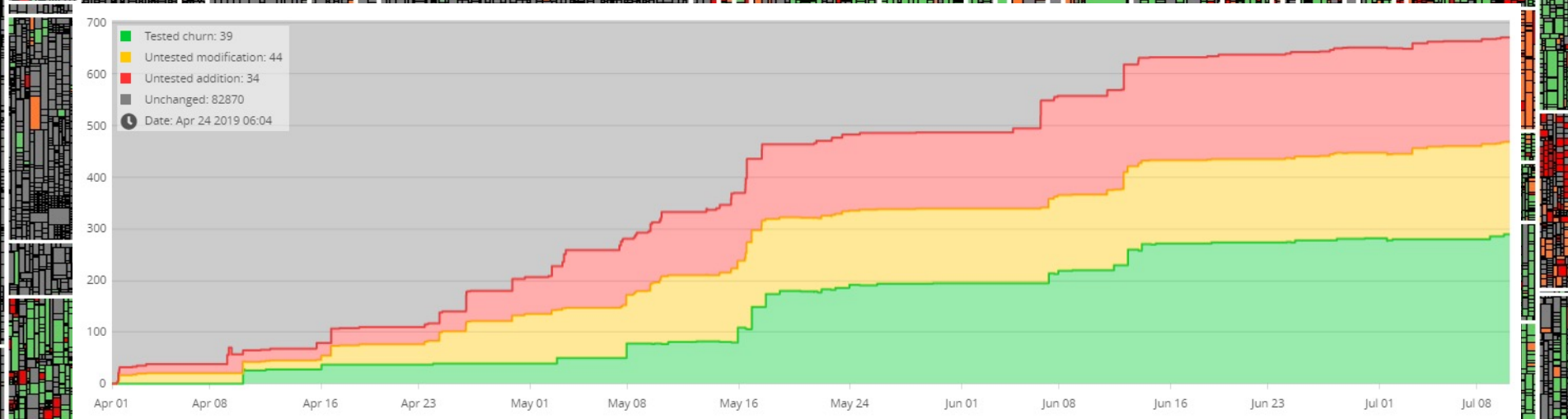
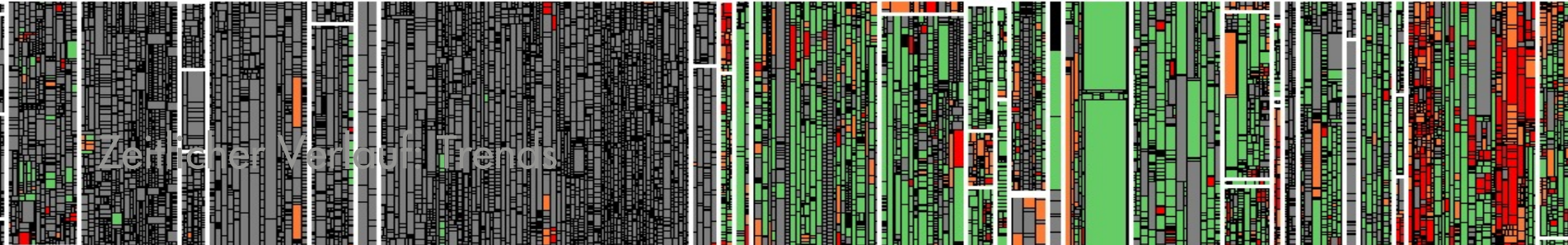
- Unverändert
- Neu & ungetestet
- Geändert & ungetestet
- Geändert & ausgeführt im Test



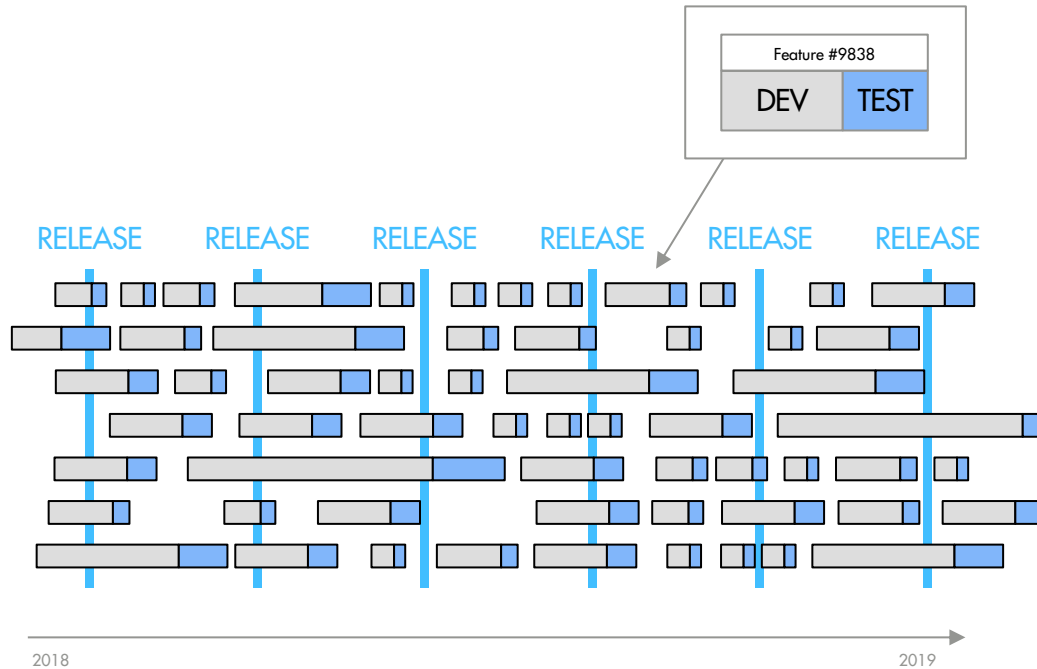
- Unverändert
- Neu & ungetestet
- Geändert & ungetestet
- Geändert & ausgeführt im Test



Zentraler Verstoß Trends



Entwicklungsbegleitender Test



Done Issue TS-23282 - clang-tidy causes SIGSEGV on C++ project (rewrite clang-tidy integration from JNI to call-in-new-process)

Updated Aug 10 2020 11:23

Creator: Nils Kunze (on May 28 2020 12:32)

Assignee: Alexander von Rhein

project	Type	Priority	Resolution	Fix Version	Component
TS	Bug	High	Green	Teamscale 6.1	Backend
Labels	Affected Version	Customer	Customer Issue	Dev Squad	Epic Name
long-runner	6.0 RC3			Denali	
Freshdesk URL	Merge Request			PDash Task	QA-Contact
	https://git.cqse.eu/cqse/teamscale/-/merge_requests/8246			#4887	wilhelm

Description

Our clang-tidy integration can lead to Teamscale crashes because the clang-tidy tool sometimes (non-deterministic) causes segfault errors. Since we execute clang-tidy via JNI in the same process as Teamscale, this segfault tears Teamscale down.

The concrete segfault appears in clang-tidy 9.0.2 (which we integrate currently) and has probably been fixed in clang-tidy 10.0.0. <https://github.com/llvm/llvm-project/commit/f28972facc1fce9589feab9803e3e8cfad01891c#diff-72222222222222222222222222222222>

[read more](#)

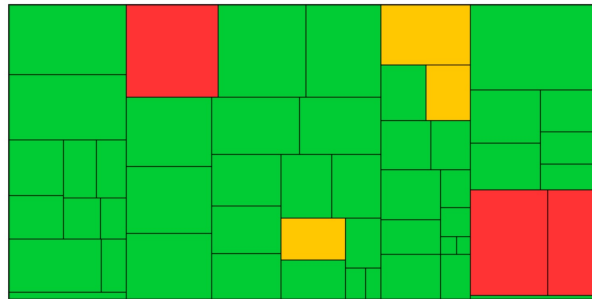
Affected files **1046**

Test Gaps

Auto-select issue branch Auto-selected: cr/23282_reimplement_clang_tidy_integration

Jun 16 2020 13:47-Now | Test Gap: 100%

Coverage sources: **All**



Findings **12** **1** **1**

Commits **44**

Issues: Bug Fix Day 9.06.20

Auto-select issue branch (Automatically selected)

▼ All issues Coverage sources: **All**

Found **210 issues** matching your query

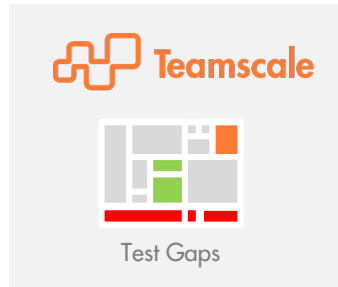
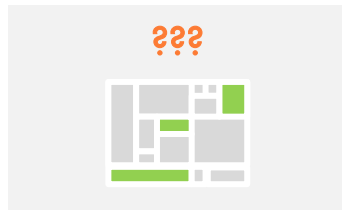
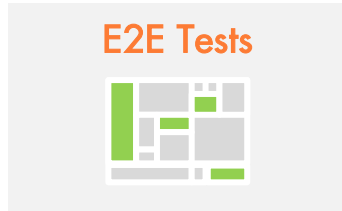
Test Gap over all matching issues: 34%

ID	Subject		# Changes	Test Gap ▼
TS-23445	GitChangeRetriever stuck in branch labeling for 10-15 minutes	Done	11	0%
TS-23460	TestImpactSynchronizer still runs OOM	Done	47	4%
TS-23547	Slow analysis progress due to long labeling	Done	8	13%
TS-23501	Security: XML External Entity vulnerability in architecture uploads	Done	7	29%
TS-23599	Potentially swallowed exception in AnalysisReportPersister	Discarded	3	33%
TS-23576	Force Rollback UI broken	Done	3	33%
TS-23446	Python architecture analysis handles late addition of __init__.py file incorrectly	Done	3	33%
TS-23450	JIRA-Integration: Duplicated Table Rows, even for the same project	Done	2	50%
TS-23458	Audit search appears to ignore line breaks	Done	3	67%
TS-23558	External Upload view doesn't load due to JSON error	Done	30	77%

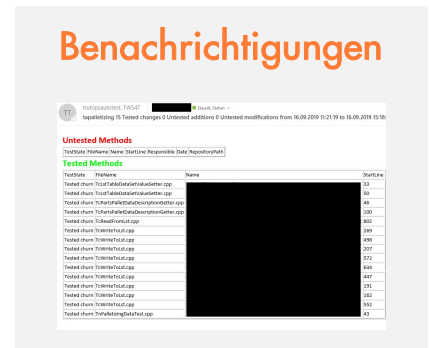
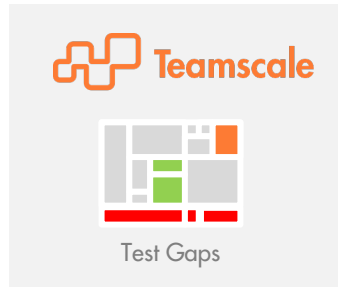
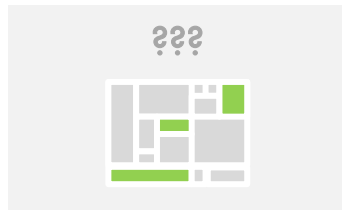
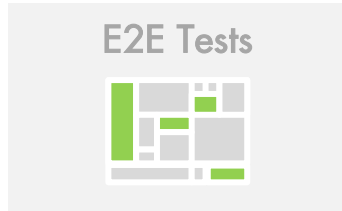
Teil 2

Herausforderungen bei der Einführung

Herausforderung: Vollständiges Bild



Herausforderung: Änderung des Entwicklungsprozesses



Herausforderung: Einfluss des Profilings



Performance



Verhalten



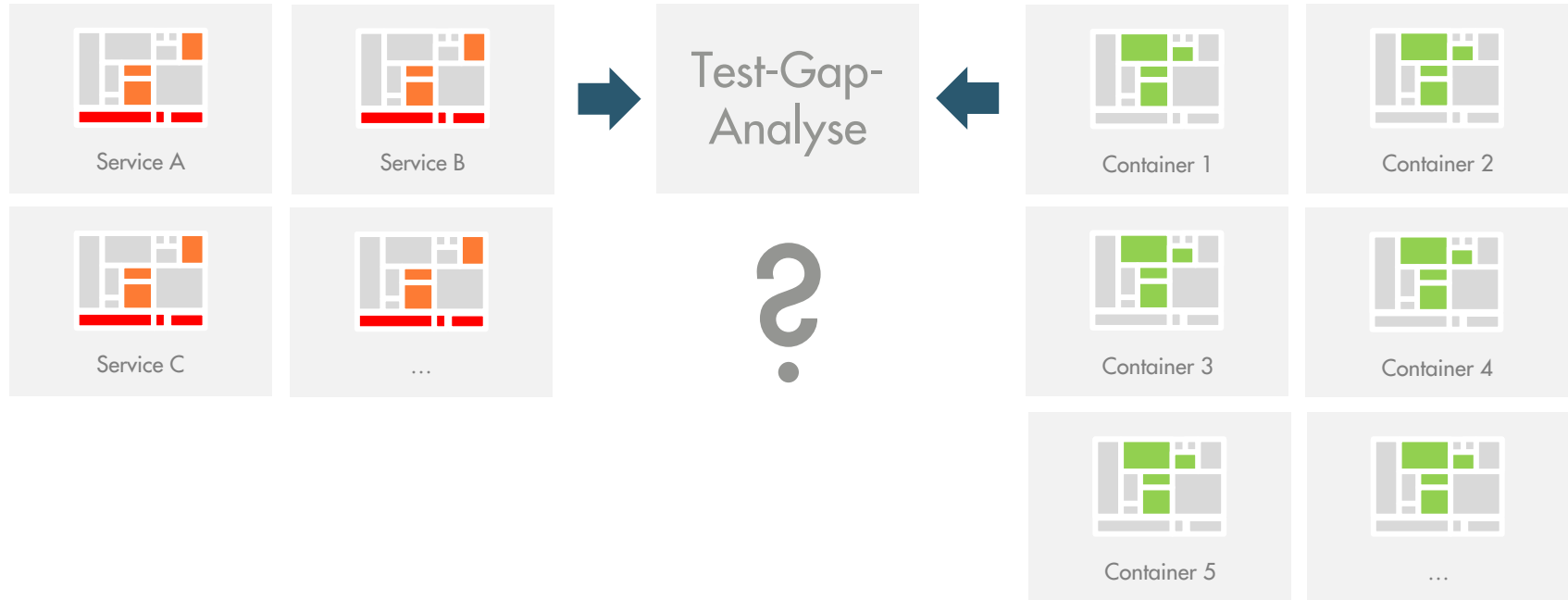
Profiler-Wahl

cqse.eu/tga-trumpf

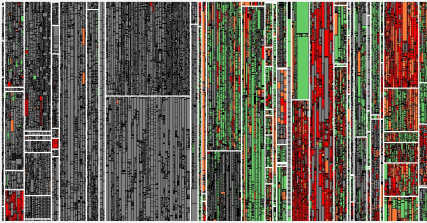


Redundanz

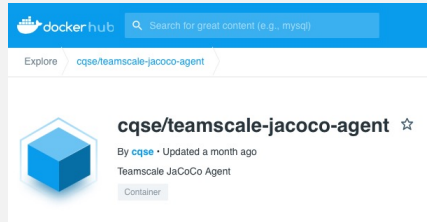
Herausforderung: Microservices



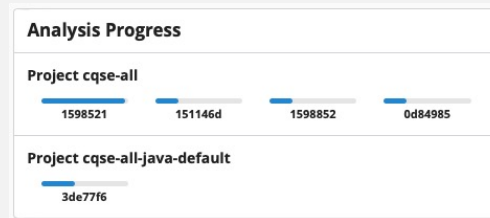
Herausforderung: Microservices



Gesamtsicht über alle
Repositories



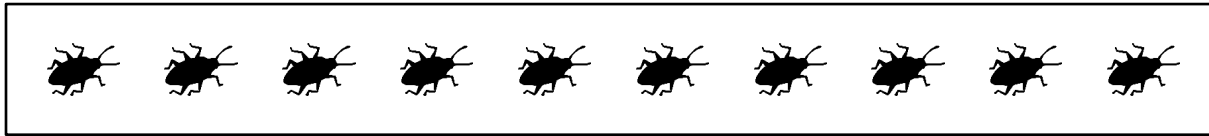
Infrastructure as Code



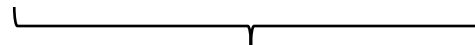
Analyse-Performance

Teil 3

Kosten-Nutzen-Berechnung der Test-Gap-Analyse

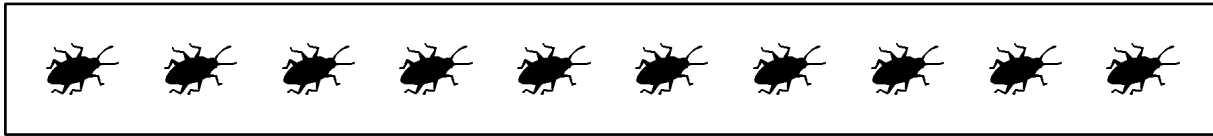


Test

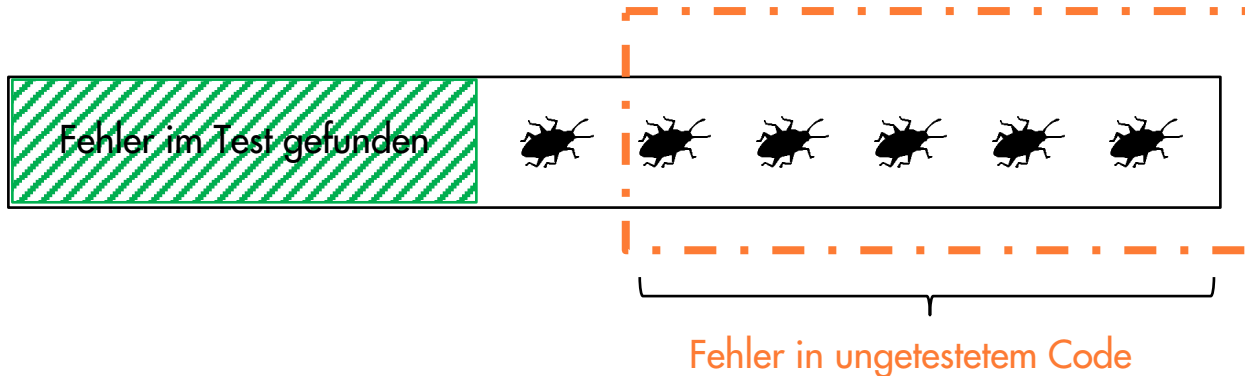


%Restfehler

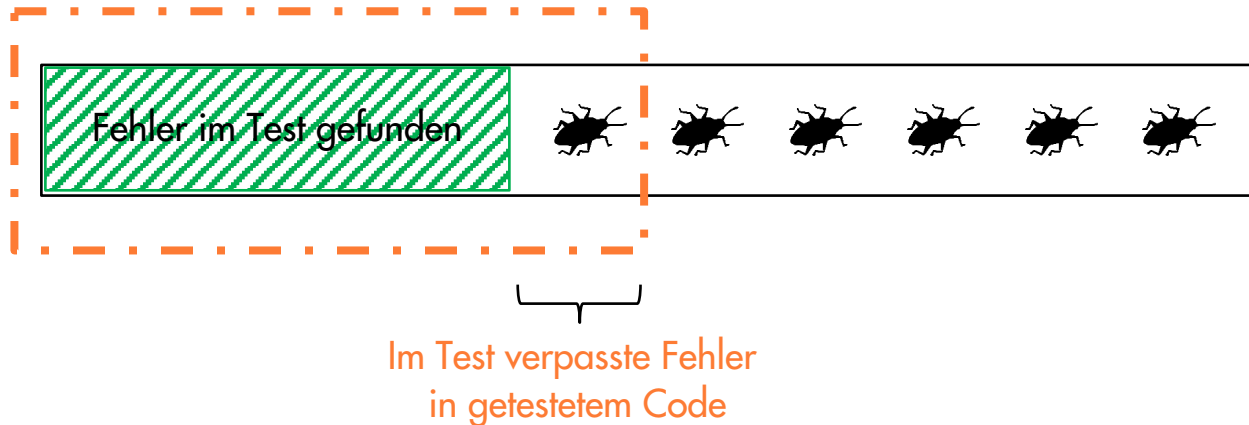
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivitat} + \% \text{Testgap}$$



Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice

Sebastian Eder, Benedikt Hauptmann,
Maximilian Junker
Technische Universität München, Germany

Einar Jørgensen
CGSE GmbH,
Germany

Rudolf Vaux, Karl-Heinz Pfrommer
Mantic Re Group,
Germany

thereof useful for maintainers and testers to identify relevant bugs in their test coverage.

B. Study Object

We perform the study on a business information system at Mantic Re. The analyzed system was written in C# and its size are 340 KiLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years and is still actively used and maintained. Therefore, there is a well implemented bug tracking and testing strategy. This allows us to gain precise data about which parts of the system were changed and why they were changed.

We analyzed two consecutive releases of the system. Release 1 was developed in five iterations in two months, and release 2 was developed in ten iterations in four months. Both releases were deployed to the productive environment due to hot fixes five times and were in productive use for six months. Thus that test deployment may concern several bugs and changes in the system. The system contained 22123 (release 1) respectively 22712 (release 2) methods.

For both releases, test suites containing 65 system test cases covering the main functionality were executed three times. A query interface that allows retrieving coverage, change, and change coverage information. The same tool support was used in earlier studies [17], [19].

Validity Procedures: We focus on validity procedures and not on threats to validity due to space limitations. We conducted manual inspections to ensure that every bug that is identified by our tool support is indeed a bug. To confirm the correctness of method genologies we build based on locally and signatures, we conducted manual inspections of randomly chosen method genologies. We found no false genologies and have therefore a high confidence in the correctness of our technique. We also used the algorithms in our former work [17], which provided suitable results as well.

C. Study Design and Execution

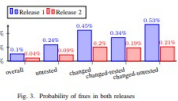
For all research questions, we classify methods according to the categories shown in Figure 2: Tested or untested, changed or unchanged, and whether methods contain field bugs.

RQ 1: Untested methods account for 34% in both releases we analyzed. 19% of all methods were changed during the development phase of the system, also in both releases. The equality of the numbers for both releases is a coincidence. 8% respectively 9% of all methods were changed/untested. Considering only changed methods, only 44% were tested in release 1 and 45% of these methods were tested in release 2. These numbers contain that there are gaps in the test coverage of changed code in the analyzed system.

RQ 2: We found 23 tests in release 1 and 10 tests in release 2. The distribution of the bugs over the different change and coverage categories of methods is shown in Table 1. The biggest part of bugs occurred in methods categorized as changed/untested with 43% of all bugs in release 1 and 40% of all bugs in release 2. In both releases, there are considerably less bugs in unchanged regions than in changed regions. The probabilities of bugs are shown in Figure 3. With 0.53% in release 1 and 0.21% in release 2, the probability of bugs is highest in the group of methods that were changed/untested. This confirms that tested code or code that was not changed in the development phase is less likely to contain field defects.

E. Discussion

RQ 1: With 15% of all methods being changed and 34% of all methods being not tested, untested code and changed code play a considerable role in the analyzed system. The high amount of changed methods results from newly developed system components, which methods that were added during the development phase of both releases.

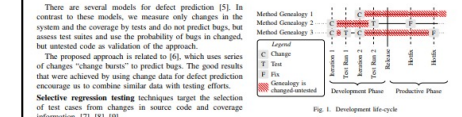


There are several models of defect prediction [5]. In contrast to these models, we measure only changes in the system and the coverage by tests and do not predict bugs, but assess test suites and use the probability of bugs as changed, but untested code as validation of the approach.

The proposed approach is related to [6], which uses changes of changes “change bins” to predict bugs. The good results that were achieved by using change data for defect prediction encourage us to combine similar data with testing efforts. **Selective regression testing** techniques target the selection of test cases from changes in source code and coverage information [7], [8], [9].

In contrast to these approaches, the paper at hand focuses on the assessment of already executed test suites, because often experts decide which tests to execute to cover most of the changes made to a software system [10]. However, their estimations contain inaccuracies and therefore possibly miss some changes. Our approach aims at identifying the resulting unexecuted code regions. Therefore, our approach can only be used if activities were already performed.

Compared to [11], we are validating our approach by measuring field defects, and do not take defects into account that were found during development. We consider a method as *tested* if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as *changed/tested*. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as *changed/untested* (see generology 1 and 3 in Figure 1).



For new features are developed. Development usually occurs in iterations which are followed by *test runs* which are the execution of a selection of test aiming to test regressions as well as the changed or added code. A development phase is completed by a *release* which transfers the system into the *productive phase*. In the productive phase, functionality is usually neither added nor changed. If critical malfunctions are detected, hot fixes are deployed in the productive phase. We consider a method as *tested* if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as *changed/tested*. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as *changed/untested* (see generology 1 and 3 in Figure 1).

Test coverage metrics: give an overview of what is covered by tests. Much research has been performed in these topics [12] and there is a plethora of tools [13] and a number of metrics available, such as statement, branch, or path coverage [14]. In contrast to these metrics, we focus on the more coarse grained method coverage. Furthermore, we do not only consider static properties of the system under test, but changes.

Empirical studies on related topics focus to the best of our knowledge mainly on the effectiveness of test case selection or prioritization techniques [9], [15]. In our study, we assess test suites by their ability to cover changes of a software system, but do not consider sub sets of test cases.

III. CONTEXT AND TERMS

In this work, we focus on *system testing* according to the definition of IEEE Std 4191-2.1996 [16] to denote “*testing conducted on a complete, integrated system to evaluate the system conformance to its specified requirements*”. System tests are often used to detect bugs in existing functionality after the system has been changed. In our context, many tests are executed manually and do not run in natural languages. Our study uses *methods* as they are known from programming languages such as Java or C#. Methods form the origin of our study and can be regarded as units of functionality of a software system. They are defined by a signature and a body of code, which differ only in the way they are executed over time. We create *method genologies* which represent the evolution of a single method over time. A genalogy connects all releases of a method and can be regarded as unit of functionality of a software system. We use the term *life cycle of a software system* composed of two alternating phases (see Figure 1). In the *development phase*, development is maintained.

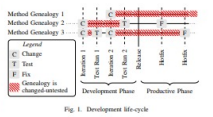


Fig. 1. Development life-cycle

change coverage is suitable for the assessment of the alignment of testing and development activities. We find that method coverage is suitable for guiding testers during the testing process. With information about change coverage, testing efforts can be assessed and reduced if necessary, because the probability of bugs is increased in changed-untested-methods. Furthermore, we presented our tool support that allows us to utilize our technique in practice. However, the number of bugs we found is too small to derive generalizable results. Therefore, we plan to extend our studies to other systems to increase external validity. But the first results that we presented in this work point out that the consideration of code regions that are modified, but not very well tested is important. This motivates future work on the topic and the inference of improvement goals.

One challenge is the identification of suitable test cases from code regions to give hints to testers and developers which test case to execute to cover more code, but untested methods. Therefore, we plan to evaluate techniques related to trace link recovery to bridge the gap to test cases.

RQ 2: With a probability of bugs in changed-methods of 0.53% respectively 0.21%, this group of methods contains most of the bugs. This means that the system itself contains few bugs at the current stage of development and bugs are brought into the system by changes. Furthermore, the probability of bugs in untested code is, in both releases, less than half of the probability in changed/untested code. Hence, we conclude that only considering test coverage is not as efficient as considering change coverage. The probability of bugs in changed code regions is also considerably higher than in untested regions. But the combination of both metrics, test coverage and changed methods points to code regions that are more likely to contain bugs than others. code regions that are more likely to contain bugs than others. code regions that are more likely to contain bugs than others.

Change Coverage Helpful in Practice? We employed the proposed approach also in the context of Mantic Re in currently running development phases. We showed the results to developers and testers by presenting code units, listing types of assemblies affected by change coverage. During the discussion of the results, we conducted open interviews with developers to gain knowledge about how helpful information about change coverage is for the maintenance and testing. Developers identified meaningful methods in changed but untested regions by using the static call graph to find methods they knew. With these methods, the developers were able to identify features that remained untested. First the processing of excel sheets in a particular calculation was changed, but remained untested afterwards. In this case, among some others, the re-execution of particular test cases and the creation of new test cases were initiated. This increased the change coverage considerably for the code regions where the features are located. This shows that change coverage is helpful for practitioners.

CONCLUSION AND FUTURES WORK We presented an automated approach to assess the alignment of test suites and changes in a simple and understandable way instead of using rather complex mechanisms to derive code units that may be subject to change. We are focusing this research question to decide whether change coverage can be used as a predictor for bugs in large code regions and if this is the case, how to use it.

Abstract—Testing and development are increasingly performed by different organizations, often in different countries and time zones. Since their complete communication, close alignment between development and testing becomes increasingly challenging. Unfortunately, poor alignment between the two threatens to decrease test effectiveness or increase costs. In this paper, we propose a conceptually simple approach to assess test alignment by answering methods that were changed but not executed during testing. The paper’s contribution is a large industrial case study that analyzes development changes, test service activity and field faults in an industrial business information system over 14 months. It demonstrates that the approach is suitable to produce meaningful data and supports alignment in practice.

Index Terms—Software testing, software maintenance, dynamic analysis, testable code

I. INTRODUCTION A substantial part of the total life cycle costs of long-lived software systems is spent on testing. In the domain of business-information systems are maintained for two or even three decades. For such systems, a substantial part of their total lifecycle costs is spent on testing to make sure that new functionality works as specified, and—equally important—that existing functionality has not been impaired.

During maintenance of these systems, test case effectiveness is crucial. Ideally, each test cycle should validate all implemented functionality. In practice, however, available resources limit each test cycle to a subset of all available test cases. Since selection of test cases for a test cycle is a non-trivial task, we suggest this selection process is central for test effectiveness.

A common strategy is to select test cases based on the change of the code since the last test cycle. The underlying assumption is that functionality that was added or changed recently more likely to contain bugs than functionality that has passed several test cycles unchanged. Empirical studies support this assumption [1], [2], [3], [4].

If development and testing efforts are not aligned well, testing might focus on code areas that did not change. This work was partially funded by the German Federal Ministry of Education and Research (BMBWF) under the sponsorship for this article by the authors.

II. RELATED WORK The proposed approach is related to the fields of defect prediction, selective regression testing, test case prioritization, and test coverage metrics. The most important difference to the named topics is the simplicity of the proposed approach and the focus on the change coverage, but not on executed sub-sets of test cases, but does not give us to improve them.

Defect prediction is related to our approach because we identify code regions that were not tested, but contained untested, with the expectation that there are more field bugs.

Selective regression testing is related to our approach because we identify code regions that were not tested, but contained untested, with the expectation that there are more field bugs.

Wo treten Fehler in Produktion auf?

Studie: C# System

Release A:

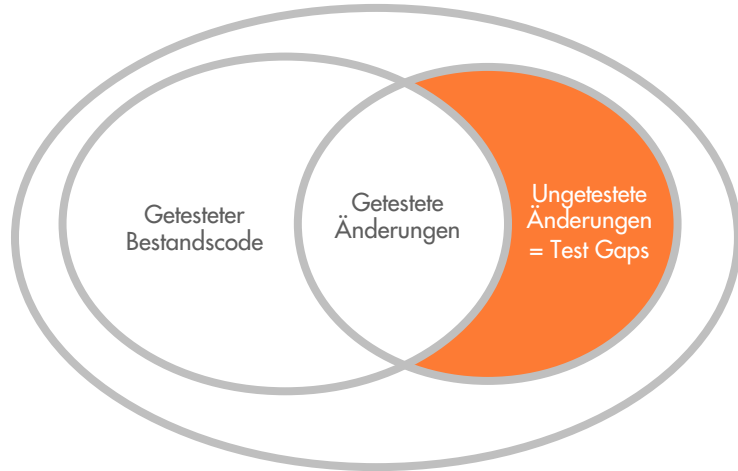
15% Code neu/geändert,

>50% ungetestet

Release B:

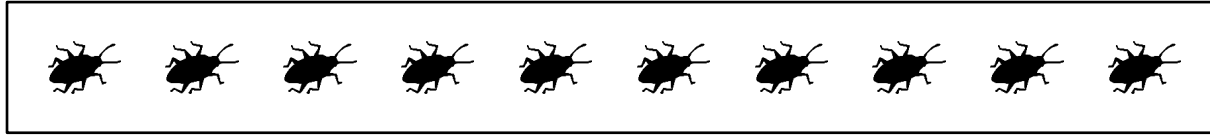
15% Code neu/geändert,

>60% ungetestet

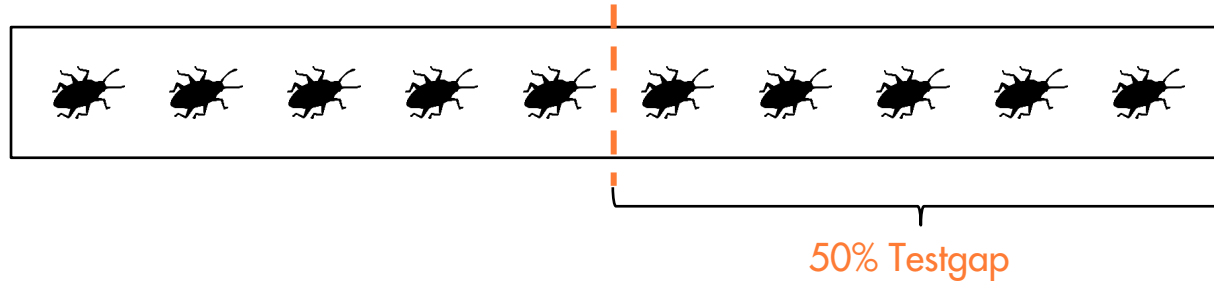


Feldfehlerwahrscheinlichkeit 5x höher für ungetestete Änderungen!

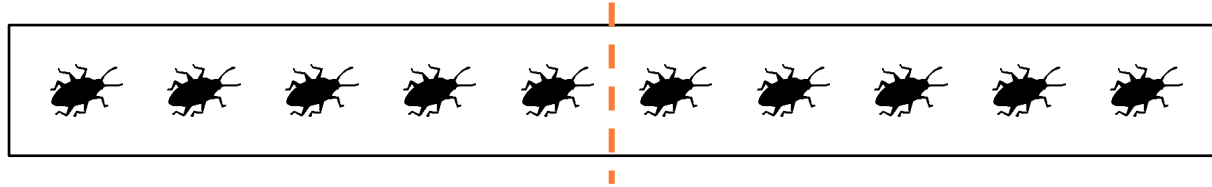
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + \% \text{Testgap}$$



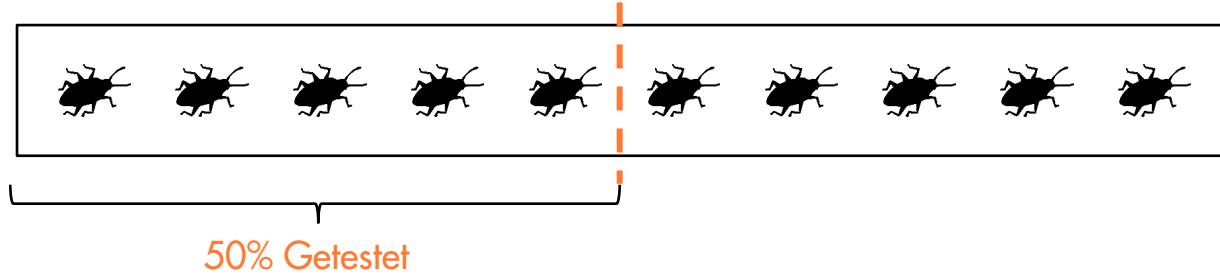
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivitat} + 50\%$$



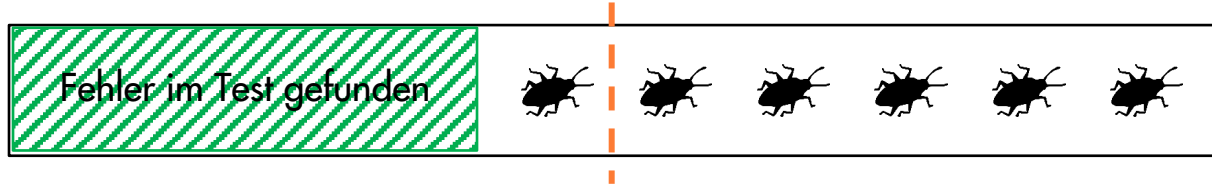
$$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivität} + 50\%$$



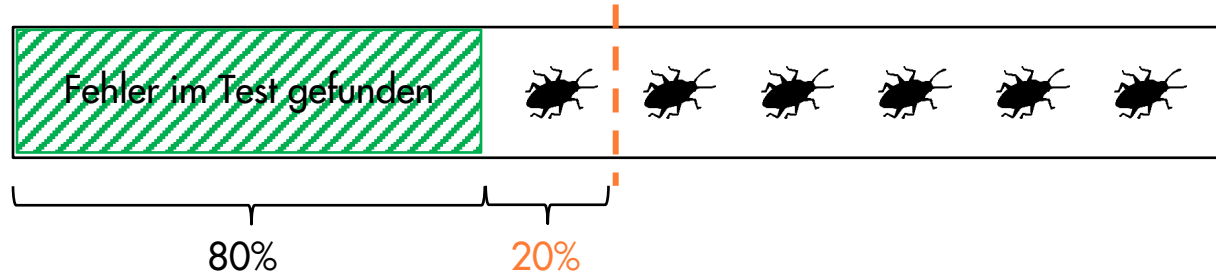
$$\% \text{Restfehler} = 50\% * \text{Testineffektivitat} + 50\%$$



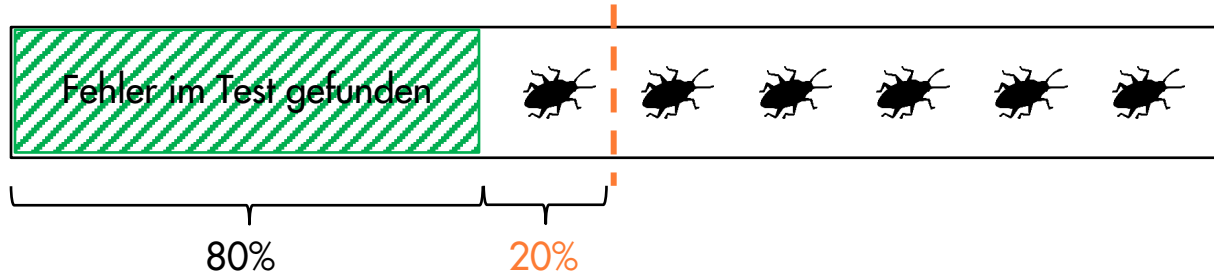
$$\% \text{Restfehler} = 50\% * \text{Testineffektivität} + 50\%$$



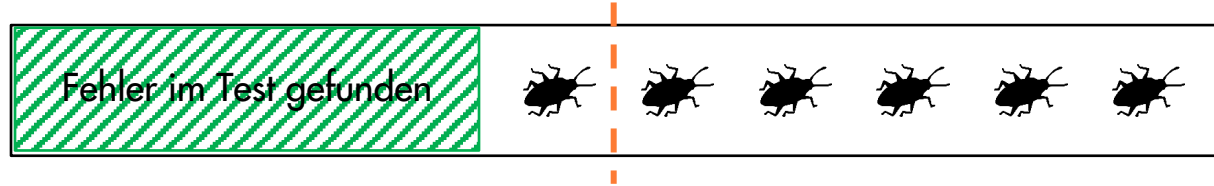
$$\% \text{Restfehler} = 50\% * \text{Testineffektivitat} + 50\%$$



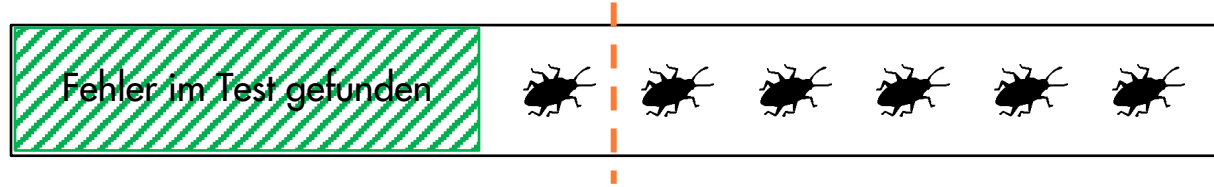
$$\% \text{Restfehler} = 50\% * 20\% + 50\%$$



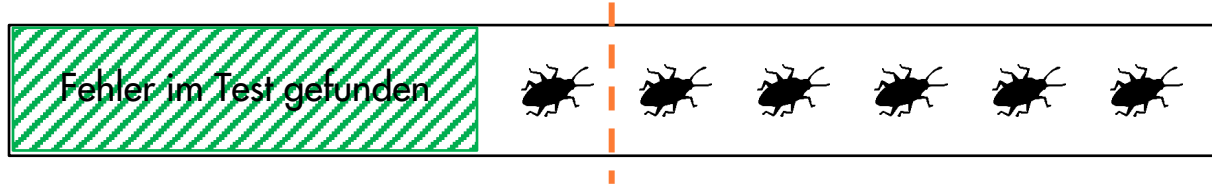
$$\% \text{Restfehler} = 10\% + 50\%$$



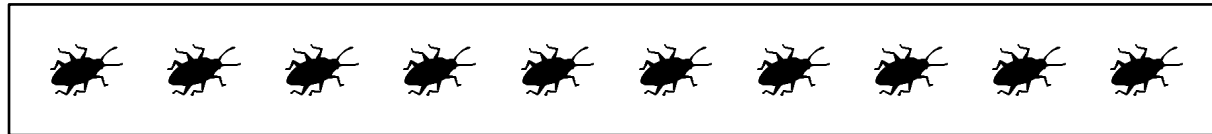
%Restfehler = 60%



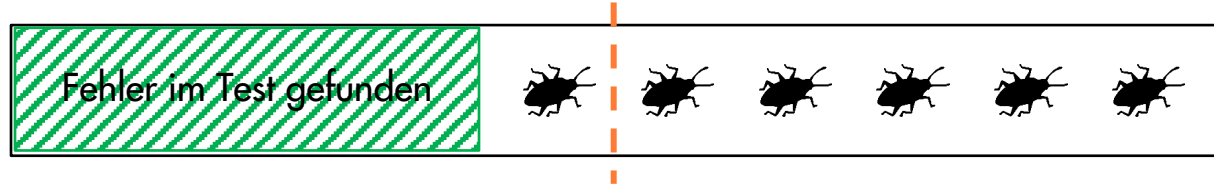
%Restfehler = 60%



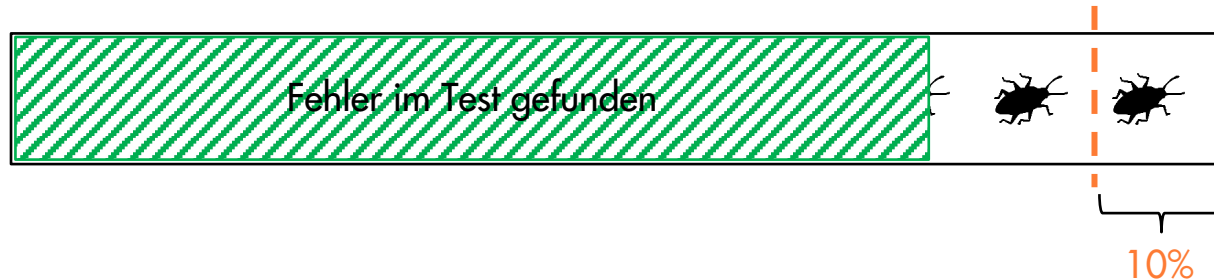
$\% \text{Restfehler} = \% \text{Getestet} * \text{Testineffektivitat} + \% \text{Testgap}$



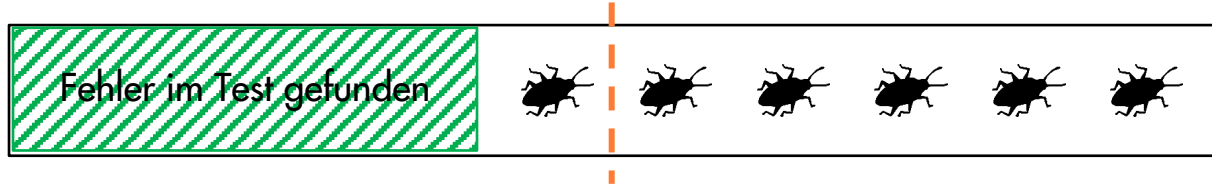
$$\% \text{Restfehler} = 60\%$$



$$\% \text{Restfehler} = 90\% * \text{Testineffektivitat} + 10\%$$



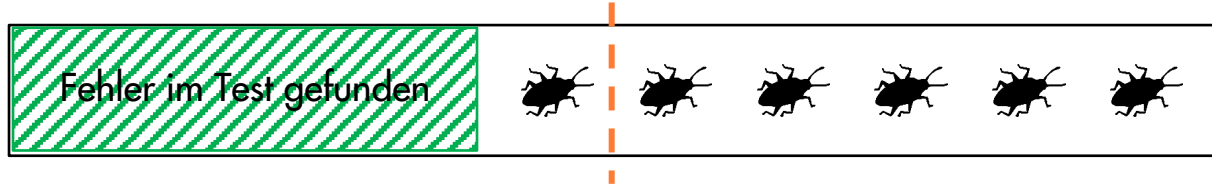
$$\% \text{Restfehler} = 60\%$$



$$\% \text{Restfehler} = 90\% * 20\% + 10\%$$



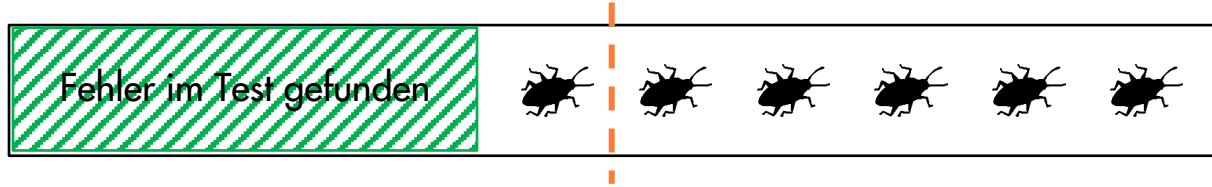
$\% \text{Restfehler} = 60\%$



$\% \text{Restfehler} = 18\% + 10\%$



$\% \text{Restfehler} = 60\%$



$\% \text{Restfehler} = 28\%$



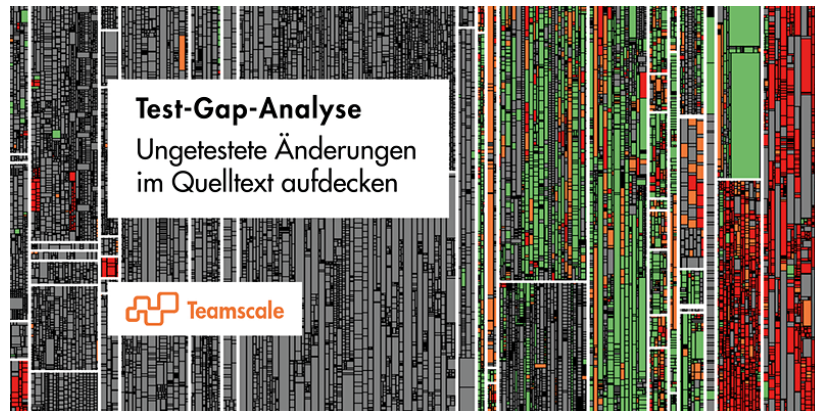
Reduzierte Feldfehler = **50%**

Test-Gap-Analyse reduziert Feldfehler in Applikationen der Munich Re um ½

Fazit

- **Sichtbarmachen** von Qualität ist essentiell
- **Werkzeuge und Prozesse** sind wichtig
- Internes **Change Management** notwendig
- Deutlicher **positiver Effekt** beobachtbar
- Am besten **gleich von Anfang** an einsetzen

Mehr Details und Live Demo im CQSE Workshop



🇩🇪 16. Februar 2022, 10:30-12:00 Uhr CEST
Registrierung: cqse.eu/tga-workshop-de-2022-02-s



🇺🇸 09. März 2022, 17:00-18:30 Uhr CEST
Registrierung: cqse.eu/tga-workshop-en-2022-03-s



Kontakt – Ich freue mich auf Diskussionen!



Dr. Sven Amann · amann@cqse.eu · +49 172 1860063

CQSE GmbH
Centa-Hafenbrädl-Str. 59
81249 München