Test-driven Perspectives for Debugging Reproducible Failures

Michael Perscheid

Software Architecture Group
Hasso-Plattner-Institut
Michael.perscheid@hpi.uni-potsdam.de

Debugging failing unit tests, particularly the search for failure causes, is often a laborious and time-consuming activity. By detecting anomalies in tested program entities, developers are able to reduce the potentially large search space. However, such anomalies do not necessarily indicate defects and so developers still have to analyze numerous candidates one by one until they find the failure cause. This procedure is inefficient since it does not take into account how suspicious entities relate to each other, whether another developer is better qualified for debugging this failure, or how erroneous behavior comes to be.

We present *test-driven perspectives* as an interconnected debugging guide that integrates anomalies and failure causes. By analyzing failure-reproducing unit tests, we reveal suspicious system parts, identify the most qualified developers, shorten the infection chain, and present erroneous behavior in the execution history and code base. Furthermore, we provide test quality feedback to recognize new failures as soon as possible. All test-driven perspectives are realized in the Paths tool suite: PathMap as an extended test runner supports a breadth first search for narrowing down failure causes, recommends developers for help, and provides test quality feedback; PathBrowser extends the source code editor with test information and automatically generated contracts from sane behavior; PathFinder is a lightweight back-in-time debugger that classifies failing test behavior for easily following infection chains back to defects; and PathView presents selected classes, methods, and their relations with respect to anomalies and failures.

1 Introduction

Debugging failing test cases tends to be a tedious activity since it requires deep knowledge of the system and its behavior [14]. For localizing failure causes, developers have to follow the infection chain backwards from the observable failure to the past defect [15]. In practice, however, this process is only partially supported by tools such as symbolic debuggers and test runners. They do not provide advice to failure-inducing anomalies, qualified developers for help, suspicious source code entities, or back-in-time capabilities. For this reason, debugging often requires too much time because developers have to rely primarily on their intuition.

To decrease the required debugging effort, techniques such as spectrum-based fault localization [6] reveal anomalies that may help in identifying failure causes. By

comparing covered program statements of passed and failed test cases, the technique produces a prioritized list of suspicious statements which restricts the search space and reduces speculations. Unfortunately, anomalies are not failure causes—developers have only a few starting points that must be debugged one by one. By debugging these suspicious source code entities, anomalies and failure causes are not integrated with each other and thus lead to a number of questions that are difficult to answer: what are the relations between anomalies and failures; who has the most experience for understanding the failure and its anomalies; which anomalies bring failures closer to defects; how are infected state and anomalous behavior propagated so that failures come to be; which classes and methods deal with anomalies and failures.

In this paper, *test-driven perspectives* integrate anomalies and failure causes to support developers in debugging reproducible failures. Developers isolate suspicious system parts, identify most qualified colleagues for help, come closer to failure causes by automatically shorten the infection chain, debug erroneous behavior back in time, and fix defects with comprehensive code views that highlight all failure-relevant program entities. Furthermore, we give developers feedback about the effectivity and efficiency of their test base to reveal new failures as soon as possible. Based on unit tests as descriptions of reproducible failures, we reveal anomalies with techniques such as spectrum-based fault localization [6] and type harvesting [3] and combine them with a compact system overview, development information, design by contract, the execution history, and the code base. The Path tool suite realizes our approach and consists of PathMap as an extended test runner, PathBrowser as a test-driven source code editor, PathFinder as a lightweight back-in-time debugger, and PathView as an interactive perspective on multiple program entities. To achieve fast response time of our tools, we base our implementation on test cases as entry points into behavior [13] and our step-wise run-time analysis [11]. Our analysis automatically splits and distributes, depending on developers needs, the dynamic analysis over multiple runs and so ensures a high degree of scalability. Thus, we answer quickly where to start debugging, who understands failures best, when defects are able to manifest for the first time, what happened before failures, which source code is affected by failures, and how new failures can be recognized more easily.

The remainder of this paper is organized as follows: Section 2 explains contemporary challenges in testing and debugging. Sections 3 describes our test-driven perspectives and how they support debugging of reproducible failures. Section 4 concludes and presents next steps.

2 Finding Causes of Reproducible Failures

Debugging is the tedious search for failure causes through the time and space of programs. Beginning with failure-reproducing behavior, in the best case in the form of test cases, developers have to follow observable failures via the infection chain back to responsible defects [15]. For instance, the small example in Figure 1 (a) illustrates that developers have to decide at each executed method what the corrupted state or behavior is so that they can trace the observable failure (bottom left) back to the defect (top). However, symbolic debuggers and standard testing frameworks face several

challenges with respect to localizing failure causes. On the one hand, debugging tools often suffer from missing advice to causes and back-in-time capabilities. Thus, developers have to rely primarily on their intuition as it is not clear how erroneous behavior comes to be. On the other hand, testing often only verifies if a failure occurs or not. There is no additional information about failure causes or at least similarities between failing and passing tests.

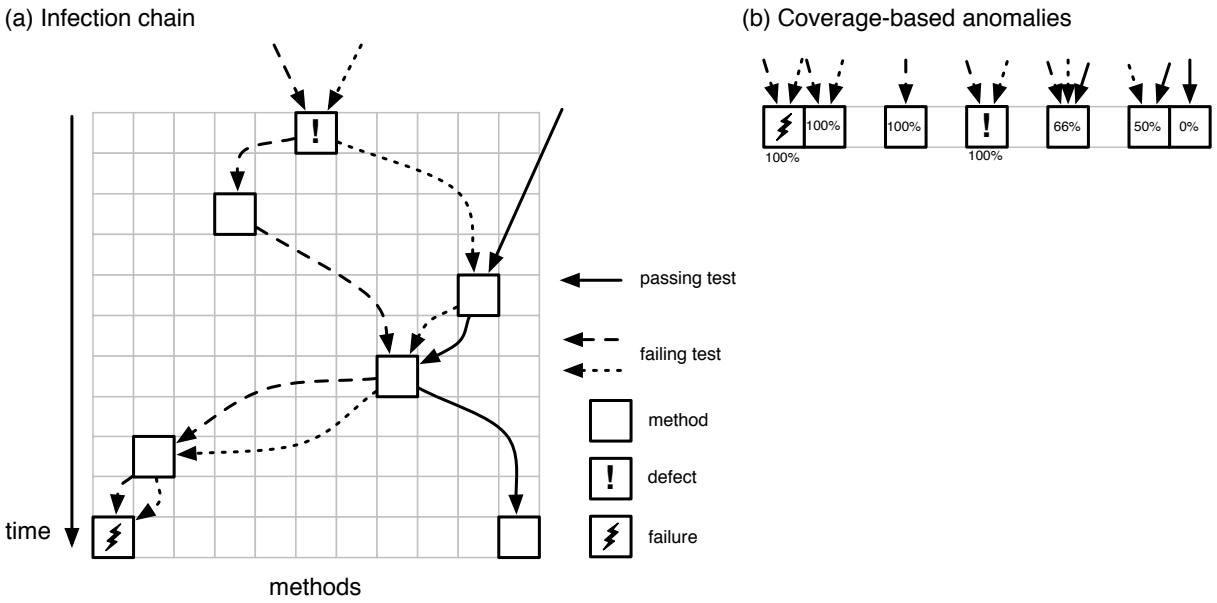

Figure 1: Developers follow the infection chain from failures back to defects or they debug a list of suspicious methods one by one.

The differences between passing and failing test cases restrict the search space for debugging. Analysis techniques that compare test cases are able to reveal anomalies in the behavior and state of programs. Such anomalies differ from expectations and they identify suspicious program entities that could be responsible for failure causes. For instance, the numbers in Figure 1 (b) reflect the suspiciousness scores from spectrum-based fault localization [6]. Based on the number of failing and passing tests per covered method a percentage value determines the failure cause probability. Although anomalies are excellent hints for starting debugging—they are neither failure causes nor defects. Since existing techniques do not consider the infection chain at all, developers have to debug a list of anomalies one by one. In our example, there are four methods with the same high suspiciousness value and it is not clear how they are related to failure causes, to each other, and the still unknown defect.

To further reduce debugging costs, we argue that it is important to integrate anomalies and failure causes. Linked views between suspicious source code entities and erroneous behavior help to localize causes more efficiently. An integration supports developers not only in answering more difficult questions but also allows other debugging activities such as the identification of experts to benefit even from anomalies.

3 Test-driven Perspectives: Integration of Anomalies and Failure Causes for Debugging Unit Tests

Test-driven perspectives integrate anomalies and failure causes and provide a comprehensive tool suite for debugging reproducible failures. Based on unit tests as entry points into failing behavior [13], developers are able to follow infection chains more easily and to investigate failure causes from different point of views. Starting from a breadth first search, they refine their understanding of corrupted program entities until they are able to fix the defect. In doing so, anomalies help to focus on failure causes and to navigate developers through the large search space of programs.

3.1 Localizing Suspicious System Parts

Having at least one failing unit test, developers can compare its execution with other test cases and identify starting points for debugging. For supporting such a breadth-first search, we provide a complete system overview that highlights relationships between anomalies and problematic areas for potential failure causes [10]. By applying and improving spectrum-based fault localization [6], which predicts anomalies by the ratio of failed and passed tests at covered methods, we automatically localize possible failure causes within a few suspicious methods and so significantly reduce the necessary search space.

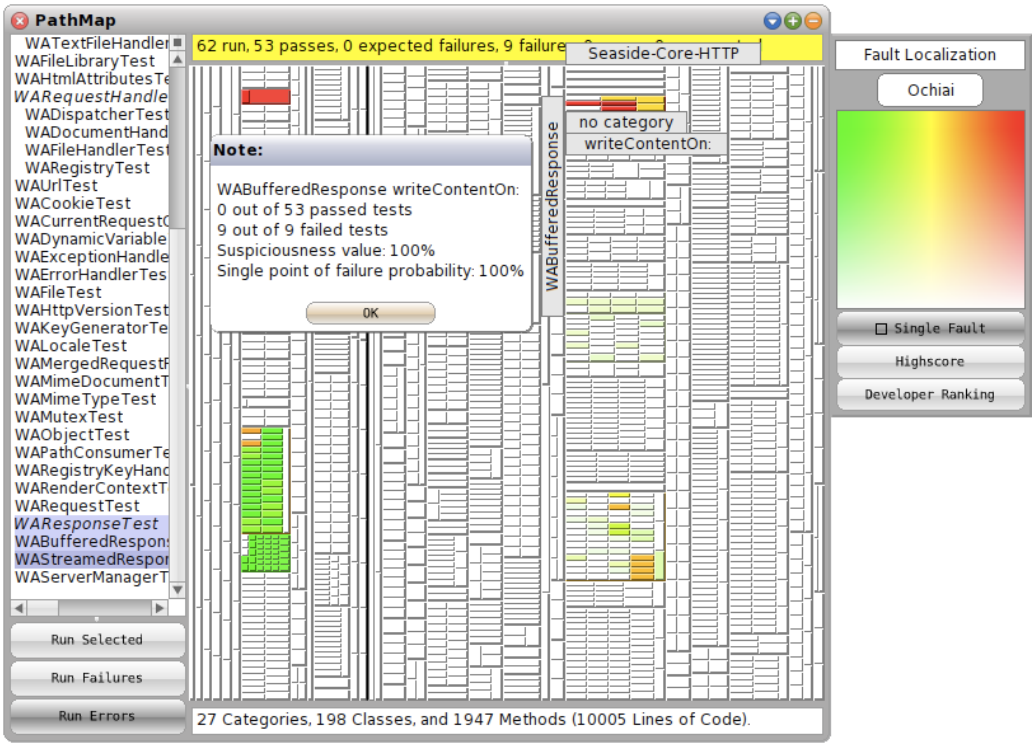

Figure 2: PathMap is our extended test runner that analyzes test case behavior and visualizes suspicious methods of the system under observation.

Our PathMap tool [8] implements this approach as an extended test runner for the Squeak/Smalltalk development environment (Figure 2). Its integral components are a scalable visualization in form of an interactive tree map and a low overhead dynamic analysis framework for computing anomalies at methods and refining results at statements on-demand.

3.2 Identification of Experienced Developers

As understanding failure causes still requires thorough familiarity with suspicious system parts, we propose a new metric for identifying expert knowledge [10]. Especially in large projects where not everyone knows everything, an important task is to find contact persons that are able to explain erroneous behavior or even fix the bug itself [1]. Assuming that the author of the failure-inducing method is the most qualified contact person, we approximate developers that have recently worked on suspicious system parts. Our metric restricts the set of possible experts to suspicious methods only and so improves the accuracy of recommended contact persons. Based on PathMap's data, we sum up suspicious and confident methods for each developer, compute the harmonic mean for preventing outliers, and constitute the proportion to all suspicious system parts. With our ranking metric, we do not want to blame developers but rather we expect that the individual skills of experts help in comprehending and fixing failure causes more easily.

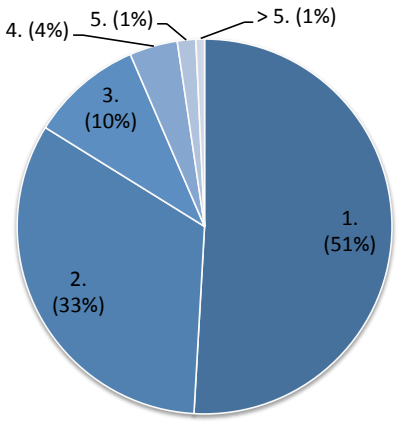

Figure 3: Ranking of the most qualified developers for 1,000 failures in Seaside.

We evaluate our developer ranking metric by introducing a considerable number of defects into the Seaside Web framework [12]. We randomly chose 1,000 covered methods and for each of them insert a defect, compute a sorted list of recommended developers, and compare it with the author of the faulty method. Seaside is an open source Web framework written by 24 developers and consists of about 400 classes, 3,700 methods and a large unit test suite with more than 650 test cases. Figure 3 illustrates the distribution of the most qualified developers and their positions in the

recommendations. For almost half of all defects, the responsible developer of the faulty method is ranked in the first place and for more than 90 % of all cases within the first three ranks. Even if developers being responsible for the fault are not listed at the top, we expect that their higher ranked colleagues are also familiar with suspiciously related system parts. Considering that failure causes are still unknown, our developer ranking metric achieves very satisfactory results with respect to the accuracy of recommended contact persons.

3.3 Generalized Contracts for Cutting the Infection Chain

To shorten entire infection chains with their numerous method calls, we introduce automatically generated contracts that identify anomalies in run-time data of failing tests. We reduce the search space of infection chains by revealing violated expectations from sane test behavior that are closer to defects. Based on dynamic contract layers [5] and context-oriented programming [2], we have a flexible extension of design by contract that allows developers to manually define valid pre-/postconditions and invariants per method. As soon as a condition is violated, an exception is thrown, another failure is visible, and the infection chain is shortened because the new failure occurs before failing test assertions. However, the manual description of expected state and behavior is not trivial, requires much time, and includes a high maintenance effort. For this reason, we automatically derive contracts from the extensive and hidden source of information of passing test cases. With the help of inductive dynamic analysis, we generalize specific run-time data and create contracts that define the proper usage of methods. For instance, TypeHarvester [3] collects type information from unit tests and all of their executed methods. Thus, wrong applied types or in other words anomalies of a failing test can be recognized as soon as possible and the gap between defects and observable failures is narrowed down.

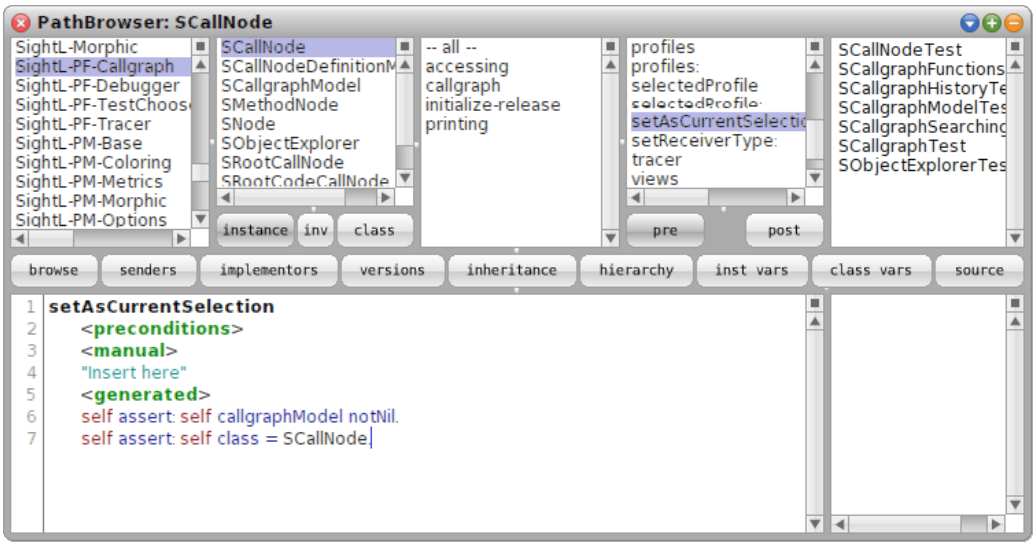

Figure 4: PathBrowser extends the source code editor with test data and automatically generated contracts from sane test behavior.

Figure 4 illustrates our extended source code editor called PathBrowser. Besides the presentation of covered unit tests per method, it allows developers to add and see contract information per method. For instance, the shown method requires a valid call graph model that was included in all passing tests. If a failing test do not have such a model, the contract is violated and an exception reveals a new observable failure. We argue that such anomalies are closer to defects because they are part of the infection chain and occur before original failures such as a failing unit test assertions.

3.4 Lightweight Back in Time Debugging of Failing Behavior

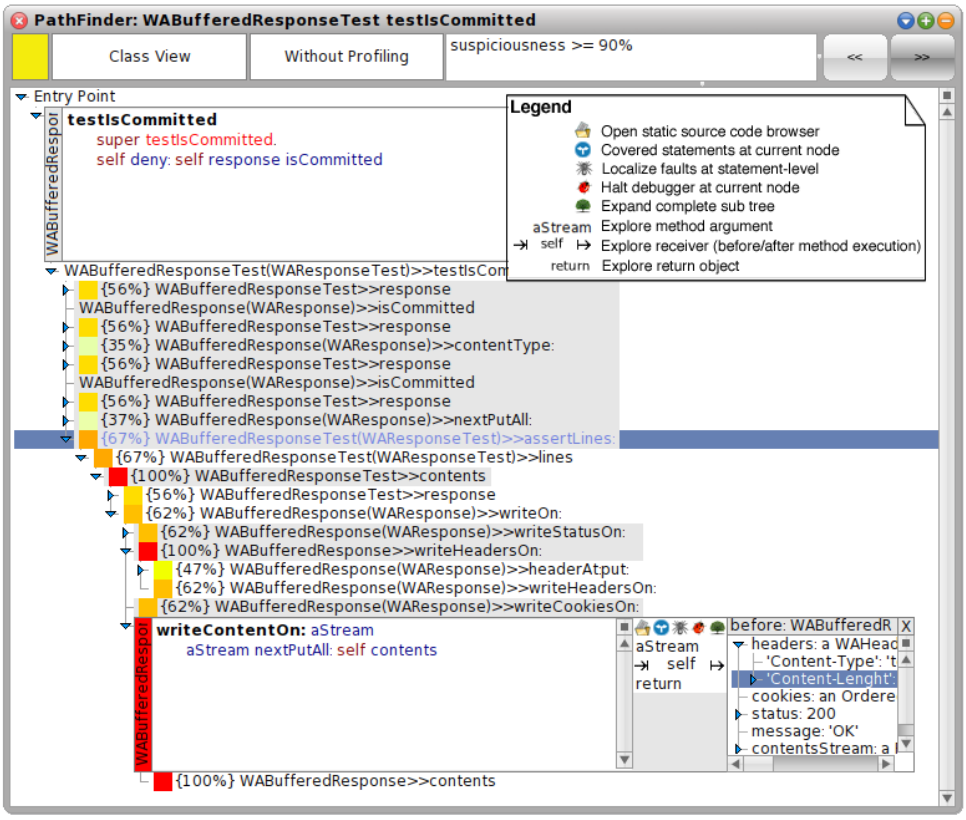

Figure 5: PathFinder is our lightweight back in time debugger that classifies failing test behavior for supporting developers in navigating to failure causes.

For understanding the relationship between anomalies and causes and to follow corrupted state and behavior back to failure-inducing origins, we offer fast access to the complete behavior of tests and their erroneous run-time data. PathFinder is our back in time debugger for introspecting specific test executions with a special focus on fault localization. It does not only provide immediate access to run-time information, but also classifies traces with suspicious methods. For localizing faults in test case behavior, developers start exploration either directly or out of covered suspicious methods as provided by PathMap. Subsequently, PathFinder opens at the chosen method as shown in Figure 5 and allows for following the infection chain back to the

failure cause. We provide arbitrary navigation through method call trees and their state spaces. Besides common back in time features such as a query engine for getting a deeper understanding of what happened, our PathFinder possesses three distinguishing characteristics. First, step-wise run-time analysis [11] allows for immediate access to run-time information of test cases. Second, the classification of suspicious trace data facilitates navigation in large traces [10]. Third, refining fault localization at the statement level reveals further details for identifying failure causes.

3.5 Anomalous Code Views for Fixing Defects

To fix defects, we provide an interactive source code view that highlights the impact of failures. Developers are able to focus on involved program entities and their relationships so that they understand how defects propagate and how to prevent similar problems. Our anomalous code views reflect classes, methods, and anomalies being involved in execution traces of failing tests. In an interactive and UML-like perspective, developers fix defects and check their influence on other application parts.

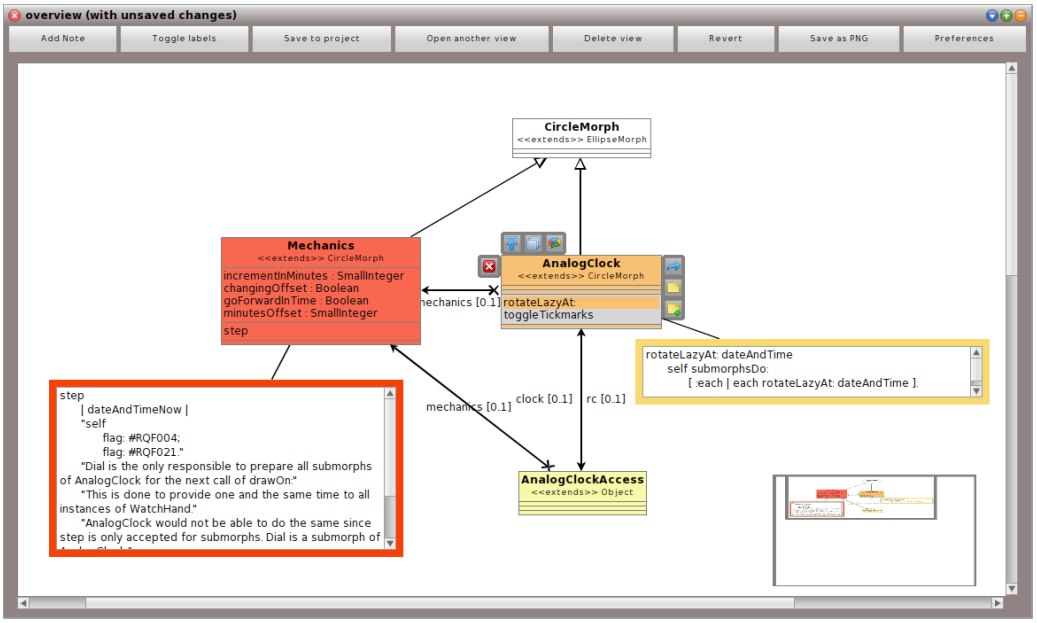

Figure 6: PathView focuses on source code entities that are influenced by failures.

Figure 6 presents our PathView tool that combines the results of spectrum-based fault localization, harvested type information [3], and a number of other reverse engineering techniques. The view is based on the most suspicious classes and methods of a failing test. The relationships between classes such as associations are automatically created with collected types. All other information is created by following a bottom up reverse engineering approach meaning that each element has a counterpart in the code base. Thus, developers can modify and extend the view and all changes are also done in corresponding source code entities. After fixing the defect, they can run the tests again and see how anomalies with their representing colors are gone.

3.6 Improved Failure Detection with Test Quality Feedback

Developers require feedback about the quality of their test base to ensure good coverage of a software system, to prioritize testing effort, and to recognize new failures as soon as possible. Test quality feedback [9] supports developers in the identification and correction of inadequately tested system parts with respect to their effectivity and efficiency. Our approach combines multiple high-level views of system, coverage, and profiling information. Effectivity feedback reveals where to increase the testing effort and who is best able to write new tests. For instance, we identify complex code with too less tests and methods that are used in reality but not covered by unit tests [4, 7]. Efficiency feedback reports on run-time performance and memory consumption to speed up the execution of entire test suites with limited effort. To ensure automated, scalable, and fast responses to developers we base our approach on our PathMap and its coverage framework that provides immediate feedback at the method level and more detailed on-demand feedback at the statement level.

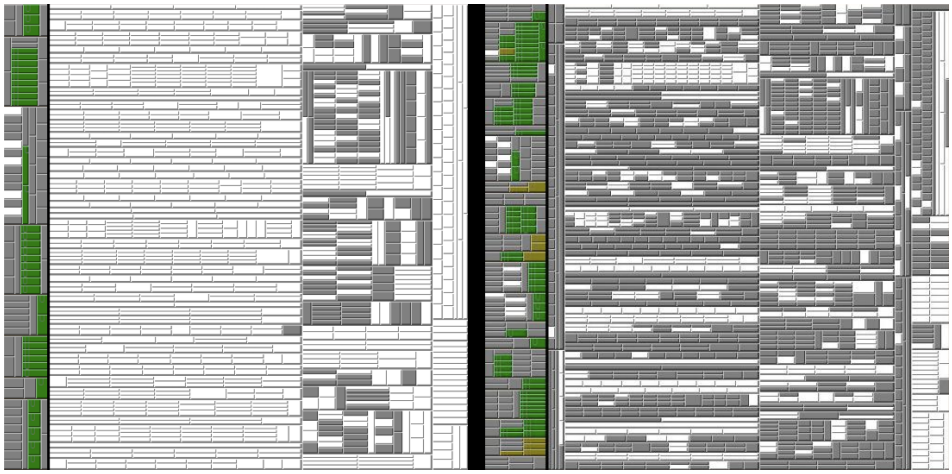

Figure 7: 4Conferences before and after the second phase of the implementation which used our test quality feedback.

We evaluate the practicality of our approach through the study of a conference management web application. The project was developed in two phases by two distinct groups of students in the context of a software engineering course. The first phase resulted in a working system with basic features only and consisted of 5 packages, 77 classes, 1126 methods, and 21.58 % coverage by unit tests. For the second phase, we asked a group of 16 bachelor students to add a specified set of features to the system. After 3 months the system resulted in 7 packages, 131 classes, and 1813 methods. As a result of their work the method coverage increased from 21.58 % in the first phase to 69.33 % at the end of the second. Figure 7 shows two tree maps of our PathMap tool. On the left-hand side the visualization shows the system after the first phase: students have only tested the model package. On the right-hand side the tree map shows the system after the second phase. The students told us they made extensive use of the test runner and especially the effectivity feedback. They told us the test runner helped them to find gaps in the coverage and made it clear which new tests they had to write.

4 Summary and Next Steps

We propose test-driven perspectives and its accompanying tool suite for simplifying debugging of failures that are reproducible via unit tests. With the help of the Path tool suite, developers can localize suspicious system parts, learn about other developers who are likely able to help, automatically shorten the infection chain by violated contracts, debug erroneous behavior back to failure-inducing origins, and fix the failure with a focused code view. Furthermore, we provide test quality feedback to strengthen the test base so that new failures can be recognized as soon as possible.

Future work is two-fold. First, dynamic contract layers for Smalltalk and our anomalous code views are still under development and need to be completed. Second, we are planning a larger user study to assess how all test-driven perspectives help and improve debugging in general.

References

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who Should Fix this Bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370. ACM, 2006.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-Oriented Programming Languages. In *COP '09: In Proceedings of the First International Workshop on Context-Oriented Programming, co-located with ECOOP*, pages 1–6. ACM DL, July 2009.

[3] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. Type Harvesting A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. In *SAC '11: Proceedings of the 25th Symposium on Applied Computing*, pages 1282–1289. ACM, March 2011.

[4] Robert Hirschfeld, Michael Perscheid, and Michael Haupt. Explicit Use-case Representation in Object-oriented Programming Languages. In *DLS '11: Proceedings of the Dynamic Languages Symposium, co-located with the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, page to appear. ACM, October 2011.

[5] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *SAC '10: Proceedings of the 25th Symposium on Applied Computing*, pages 2169–2175. ACM, March 2010.

[6] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM, 2002.

[7] Michael Perscheid. Requirements Traceability in Service-oriented Computing. In *Proceedings of the Fall 2009 Workshop of the HPI Research School on Service-Oriented Systems Engineering*. Hasso-Plattner-Institut, October 2009.

[8] Michael Perscheid. Understanding Service Implementations Through Behavioral Examples. In *Proceedings of the Fall 2010 Workshop of the HPI Research School on Service-Oriented Systems Engineering*. Hasso-Plattner-Institut, October 2010.

[9] Michael Perscheid, Damien Cassou, and Robert Hirschfeld. Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing. In *C5 '12: Proceedings of the Conference on Creating, Connecting and Collaborating through Computing*, page accepted, January 2012.

[10] Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. In *JSSST '11: Proceedings of the JSSST Annual Conference*, September 2011.

[11] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*, pages 77–86. IEEE, October 2010.

[12] Michael Perscheid, David Tibbe, Martin Beck, Stefan Berger, Peter Osburg, Jeff Eastman, Michael Haupt, and Robert Hirschfeld. *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008.

[13] Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into Examples: Leveraging Tests for Program Comprehension. In *TestCom 2009: Proceedings of the 21st IFIP International Conference on Testing of Communicating Systems*. Springer-Verlag, November 2009.

[14] Iris Vessey. Expertise in Debugging Computer Programs: A Process Analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[15] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2006.