In Proceedings of the JSSST Annual Conference 2011,
Naha, Okinawa, Japan, September 27-29, 2011.

Test-Driven Fault Navigation for Debugging Reproducible Failures

Michael Perscheid, Michael Haupt, Robert Hirschfeld and
Hidehiko Masuhara

Debugging activities, particularly those for searching for failure causes, are often laborious and time-consuming. Techniques such as spectrum-based fault localization or back-in-time debugging help programmers to reduce development cost. However, such approaches are often limited to a single point of view, ignoring the need for combined perspectives.

We present *test-driven fault navigation* as an interconnected guide to failure causes. Based on failure-reproducing unit tests, we introduce a novel systematic top-down debugging process with corresponding tool support. With spectrum-based fault localization, we offer navigation to suspicious system parts and erroneous behavior in the execution history and rank developers most qualified for addressing the faults localized. Our evaluation illustrates the practicability of this approach, its high accuracy of developer recommendation, and the fast response times of its corresponding tool suite.

1 Introduction

Debugging is one of the most laborious development activities. The search for failure causes requires deep knowledge of the system and its behavior [26]. Developers have to follow the infection chain backwards from the observable failure to the past defect [27]. In practice, this process is mostly manual and tedious since standard debuggers offer neither advice to failure-inducing origins nor back-in-time capabilities. Thus, this activity often consumes a significant amount of time.

To decrease the required effort for localizing failure causes, researchers have proposed several techniques which, however, only provide either a purely static, dynamic, or expertise-focused point of view. *Spectrum-based fault localization* [9] produces a prioritized list of suspicious statements from the dynamic analysis of test cases. These source code snippets have no relations to erroneous behavior and it is not clear how failures come to be or how infected state is propagated. *Back-in-time debuggers* [12] focus on dynamic views of execution history. Unfortunately, the missing classification of suspicious or harmless behavior forces developers to make several and often laborious decisions on which execution subtree to follow. *Automatically assigning bug reports* to more experienced developers [2] reduces overall debugging time, but the problem of how to identify failure causes remains.

Although all of these techniques have respective benefits, we expect their combination to be able to limit the shortcomings. We argue that efficient debugging requires linked views between suspicious source code entities and erroneous behavior, as well

---

再現可能な誤りをデバッグするためのテスト駆動型誤り発見ツール

Michael Perscheid, Michael Haupt, and Robert Hirschfeld, Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany.
増原英彦, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo.

as qualified developers for analyzing these entities.

In this paper, we present *test-driven fault navigation* as a systematic top-down debugging process with corresponding tool support that guides developers to failure causes within structure and behavior, and also to corresponding expert members of the development team. Developers can localize suspicious system parts, debug erroneous behavior back in time, and identify colleagues for help. Based on unit tests as descriptions of reproducible failures, we combine the results of spectrum-based fault localization with a compact system overview, execution history, and development information.

The *Path tool suite* realizes our approach and consists of an extended test runner (*PathMap*) for visualizing suspicious system parts, a lightweight back-in-time debugger for unit tests (*PathDebugger*), and a metric for linking failures to suitable developers. By leveraging unit tests as a basis for dynamic analysis, we can ensure a high degree of automation, scalability, and performance. Thus, we expect to further reduce the cost of debugging as we are able to answer where the failure cause is located, how erroneous behavior is related to suspicious methods, and which developer is most qualified for fixing the bug.

The contributions of this paper are as follows:

- A novel systematic debugging process for understanding and fixing faults that can be reproduced by tests.
- A test-driven fault navigation technique that suggests interconnected advice to failure causes in structure, behavior, and experts in the development team.
- A realization of our approach by providing integrated tool support for the Squeak/Smalltalk IDE.

We evaluate our approach with respect to practicability in a real-world project, accuracy of our developer ranking metric, and efficiency of our tool suite.

The remainder of this paper is structured as follows: Section 2 introduces our motivating case study and explains contemporary challenges in testing and debugging. Section 3 presents test-driven fault navigation as a systematic debugging process, followed by a detailed description of its supporting tools. Section 4 evaluates the practicability, accuracy, and efficiency of our approach. Section 5 discusses related work, and Section 6 concludes.

2 Finding Causes of Reproducible Failures

We introduce a motivating example for an error taken from the Seaside Web framework [19] that serves as a basis for our discussion of challenges in testing and debugging. Seaside is an open source Web framework written in Smalltalk and consists of about 400 classes, 3,700 methods and a large unit test suite with more than 650 test cases. By this example, we will demonstrate test-driven fault navigation in Section 3.

2.1 Typing Error Example in Seaside

We have inserted a defect into Seaside's Web server and its request/response processing logic (`WABufferedResponse` class, `writeHeadersOn:` method). Figure 1 illustrates the typing error inside the header creation of buffered responses. The typo in "Content-Lenght" is inconspicuous but leads to invalid results in requests that demand buffered responses. Streamed responses are not influenced and still work correctly.

Although the typo is simple to characterize, observing it can be laborious. First, some clients hide the failure since they are able to handle corrupted header information. Second, as the response header is built by concatenating strings, the compiler does not report an error. Third, by reading source code like a text, developers tend to overlook such small

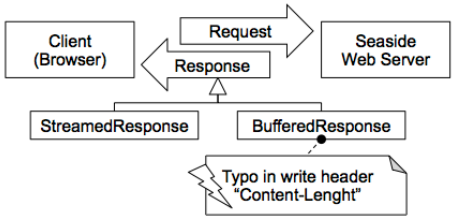

Fig. 1 An inconspicuous typo in writing buffered response headers leads to faulty results of several client requests.

typos.

2.2 Challenges in Testing and Debugging

Localizing our typing error with standard tools such as test runner and debugger can be cumbersome. Figure 2 depicts a typical debugging session. First, Seaside's test suite answers with 9 failed and 53 passed test cases for all response tests. Since all failing runs are part of `WABufferedResponseTest`, developers might expect the cause within buffered responses. However, this assumption lacks evidence, such as a list of methods being executed by all failed tests. Second, starting the standard debugger on a failing test shows a violated assertion within the test method itself. This, however, means that developers only recognize the observable failure instead of its origin. Only the current stack is available, but our typo is far away from the observable malfunction. Third, the thrown assertion suggests that something is different from the expected response. Developers have to introspect the complete response object for localizing the typo. There are no pointers to the corrupted state or its infection chain. Remarkably, the response status is still valid (`200, OK`). Furthermore, in our example we assume that developers are aware of Seaside's request/response processing. However, developers' expertise significantly influences the required debugging effort; for instance, less experienced developers need more time for comprehending the client server communication.

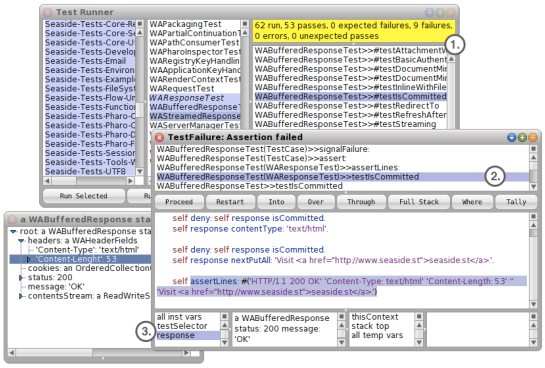

Fig. 2 Localizing failure causes with standard tools is cumbersome.

In general, standard testing and debugging approaches face several challenges with respect to localizing failure causes. Although testing is a widely adopted practice, especially in agile development [3], it 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. It is not clear how erroneous test behavior is related to each other. We expect that tests and their behavior own an extensive and hidden source of information not only for fault localization.

By starting the debugger, developers are able to introspect failed tests at the point in time where the assertion or exception was thrown. Such observable failures are in many cases far apart from the failure-inducing cause that happened in the past. For this reason, developers have to follow the infection chain backwards from the observable failure via infected state and behavior to the defect [27]. However, most debuggers do not support back-in-time capabilities, and if they do, these features often come with a performance overhead [12] or a more complicated setup [21]. Moreover, both kinds of debuggers force developers to make several de-

cisions regarding how to follow the infection chain. We argue that lightweight back-in-time debuggers as well as new navigation concepts for examining behavior can reduce the cost of debugging.

Apart from good tool support, developers' expertise is also important for the entire debugging process [2]. The required debugging effort significantly depends on individual skills and knowledge about the system under observation. More experienced developers invent better hypotheses about failure causes than novices that do not know the code base. We believe that dedicated developers understand causal relations or fix defects in less time. Unfortunately, the identification of corresponding experts is quite challenging since the observable failure does not explicitly reveal infected system parts.

3 Test-Driven Fault Navigation

We present test-driven fault navigation for debugging failures reproducible by unit tests. Our systematic top-down process and accompanying Path tools guide developers with interconnected advice to failure causes in structure, behavior, and to corresponding experts in the development team.

3.1 Debugging Reproducible Failures

Localizing non-trivial faults requires a systematic procedure to find the way in endless possibilities of time and space [27]. Experienced developers apply a promising debugging method by starting with a breadth-first search [26]. They look at a system view of the problem area, classify suspicious system parts, and refine their understanding step by step. However, independent and specialized debugging tools does not coherently support such a systematic procedure. This often leads to confusing and time-consuming debugging sessions, especially for novice developers who trust more in intuition instead of searching failure causes systematically.

We introduce a systematic top-down debugging process with corresponding tools that not only supports the method of experts but also provides guidance for novices. Developers are able to navigate from failures to causes by reproducing observable faults with the help of test cases. Afterwards, they can isolate possible defects within parts of the system, understand corresponding erroneous behavior, and optionally identify other developers for help. Figure 3 summarizes our test-driven fault navigation process and its primary activities:

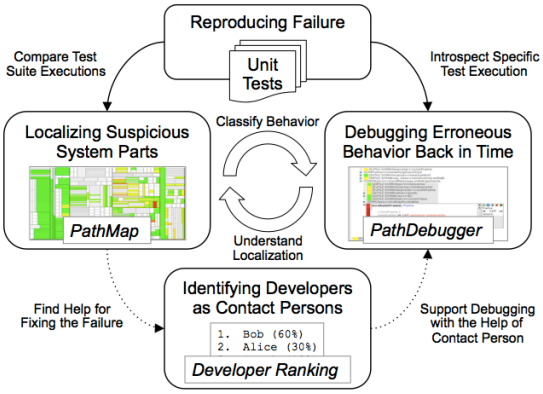

Fig. 3 Our debugging process guides with interconnected advice to reproducible failure causes in structure, behavior and to corresponding experts.

Reproducing failure

As a precondition for all following activities, developers have to reproduce the observable failure in the form of at least one unit test. Besides the beneficial verification of resolved failures, we require tests above all as entry points for analyzing erroneous behavior. We have chosen unit test frameworks because of their importance in current development projects. Our approach is neither limited to unit testing nor does it require minimal test cases as proposed by some guidelines [3].

Localizing suspicious system parts (Struc-

tural navigation)

Having at least one failing test, developers can compare its execution with other test cases and identify structural problem areas. By analyzing failed and passed test behavior, possible failure causes are automatically localized within a few suspicious methods so that the necessary search space is significantly reduced. We have developed an extended test runner called *PathMap* that provides both a static overview and related dynamic test information.

Debugging erroneous behavior back in time (Behavioral navigation)

For refining their understanding of erroneous behavior, developers explore the execution and state history of a specific test. To follow the infection chain back to the failure cause, they can start our back in time *PathDebugger* either at the failing test directly or at arbitrary methods as recommended by PathMap. If suspicious system parts are available, conspicuous methods classify the executed trace and so ease the behavioral navigation to defects.

Identifying developers as contact persons (Team navigation, optional)

Some failures require expert knowledge of others so that developers understand and debug faults more easily. By combining localized problem areas with source code management information, we provide a novel *developer ranking metric* that identifies the most qualified experts for fixing a failure. Developers having changed the most suspicious methods are more likely to be experts than authors of non-infected system parts. We have integrated our metric within PathMap providing navigation to suitable team members.

Besides our systematic process for debugging reproducible failures, the combination of unit testing and spectrum-based fault localization also provides the foundation for interconnected navigation with a high degree of automation. All activities and their results are affiliated with each other and so allow developers to explore failure causes from combined perspectives. Our tools support these points of view in a practical and scalable manner.

3.2 Localizing Suspicious System Parts

For supporting a breadth-first search, we provide a complete system overview that highlights problematic areas for potential failure causes. Applying spectrum-based fault localization, which predicts failure causes by the ratio of failed and passed tests at covered methods, we analyze overlapping test behavior, identify suspicious system parts, and visualize the results. Our PathMap tool implements this approach as an extended test runner for the Squeak/Smalltalk development environment (Figure 4). Its integral components are a compact visualization in form of an interactive tree map, a lightweight dynamic analysis framework for recording test executions, and different fault localization metrics for identifying suspicious methods.

We visualize a structural system overview and its relation to test case execution in form of a compact and scalable tree map [24]. We reflect selected categories[†1] as full columns that include their classes as rows which in turn include methods[†2] as small boxes. The allocated space is proportional to the number of methods per node. All elements are organized alphabetically, and for a clear separation we distinguish between test classes on the left-hand side and core classes on the right-hand side (label 2 in Figure 4). The entire map can interactively be explored to get more details about a specific node (label 4 in Figure 4). Furthermore, each method can be colorized with a hue element between green

†1 Categories are similar to packages in other programming languages.

†2 We provide Smalltalk's method categories as an optional layer, too.

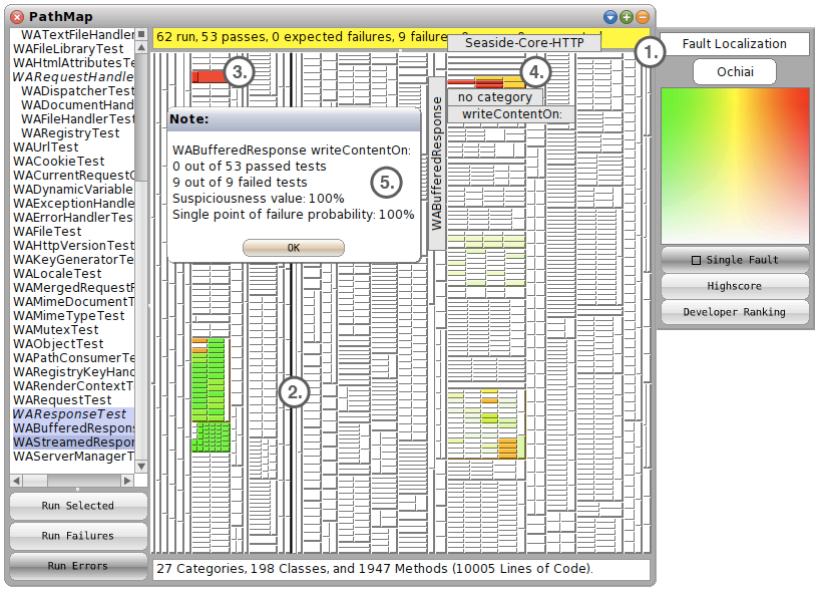

Fig. 4 PathMap is our extended test runner that analyzes test case behavior and visualizes suspicious methods of the system under observation.

and red for reflecting its suspiciousness score and a saturation element for its confidence. For instance, methods with a high failure cause probability possess a full red color. Such a visualization allows for a high information density at a minimal required space. The tree map in Figure 4 consists of only 500×500 pixels but is able to scale up to 4,000 methods. Even though this should suffice for most medium-sized applications, PathMap allows for filtering specific methods such as accessors, summarizing large elements, and resizing the entire tree map.

We ensure scalability of spectrum-based fault localization by efficiently recording test coverage with method wrappers [4]. To reduce the overhead of run-time observation, we restrict instrumentation to relevant system parts and dynamic analysis to the granularity level of methods. With the focus on selected categories we filter irrelevant code such as libraries where the defect is scarcely to be expected. Analyzing only executed methods provide a good trade-off between comprehensibility and scalability since they offer both an extensive breadth-first search and a lightweight dynamic analysis.

Based on collected test behavior, we automatically rank covered methods and visualize suspicious information in our tree map. In spectrum-based fault localization [9] failure cause probabilities are estimated by the ratio of all failing tests to test results per covered source code entity. Thus, methods are more likely to include the defect if they are executed by a high number of failing and a low number of passing tests. We distinguish between suspiciousness and confidence values of methods. While the former scores the failure cause probability with respect to covered tests and their results, the latter measures the degree of significance based on the number of all test cases. A lot of metrics for spectrum-based fault localization have been proposed among which Ochiai has shown to be the most effective one [1].

$$suspicious(m) = \frac{failed(m)}{\sqrt{totalFailed * (failed(m) + passed(m))}}$$

This formula returns a value between 0 and 1 for each method m being covered by at least one test. To visualize this result, we colorize method nodes in our tree map with a hue value between green and red. For instance, a suspiciousness score of 0.7 creates an orange area.

To assess the significance of a suspiciousness value, we apply a slightly adapted confidence metric. We only consider the relation between failed tests per method and all failing tests as we are not interested in sane behavior for fault localization.

$$confidence(m) = \frac{failed(m)}{totalFailed}$$

The returned value is directly mapped to the saturation component of already colorized method nodes. By looking only at faulty entities, we reduce the visual clutter of too many colors and results. For instance, a method covered by three out of six failing tests is grayed out.

Adapting spectrum-based fault localization to unit testing limits the influence of multiple faults. The effectiveness of existing spectrum-based approaches suffers from overlapping test cases describing different failures as well as coincidentally correct test cases which execute failures but do not verify their appearance. The selection of suitable unit test suites allows for ignoring such problematic tests and to focus on a single point of failure. Furthermore, based on the origination condition of single faults [23], which means each failure must evaluate the defect, PathMap optionally filters methods which were not executed by all failing tests. Thus, developers choose designated test suites, further reduce fault localization results, and concentrate on one specific failure at a time.

In our motivating typing error, PathMap localizes the failure cause within a few methods of Seaside's response classes. In Figure 4, developers only execute the response test suites as in ordinary test runners with the result of 53 passed and 9 failed tests (1). In the middle (2) they see a tree map of Seaside's structure with test classes on the left side and core classes on the right side[†3]. Each color represents the suspiciousness score of a method revealing problem areas of the system. For instance, the interactively explorable red box (3) illustrates that all nine failing tests are part of the buffered test suite. In contrast, the green box below includes the passed streaming tests and in orange shared test methods. The more important information for localizing the failure cause is visualized at (4). There are three red and orange methods providing confidence that the failure is included in the `WABufferedResponse` class. To that effect, the search space is reduced to six methods. However, a detailed investigation of the `writeContentOn:` method (5) shows that it shares the same characteristics as our failure cause, i. e., `writeHeadersOn:`. At this point, it is not clear from a static point of view how these suspicious methods are related to each other. Developers need an alternative view of failing test behavior in order to understand how the failure comes to be.

3. 3 Debugging Erroneous Behavior Back in Time

To follow corrupted state and behavior back to failure-inducing origins, we offer fast access to failing tests and their erroneous run-time data. Based on our lightweight PathFinder tool [18], PathDebugger 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 local-

†3 For the purpose of clarity, we limit the partial trace to Seaside's core.

izing faults in test case behavior, developers start exploration either directly or out of covered suspicious methods as provided by PathMap. Subsequently, PathDebugger 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 PathDebugger possesses three distinguishing characteristics. First, step-wise run-time analysis allows for immediate access to run-time information of test cases. Second, the classification of suspicious trace data facilitates navigation in large traces. Third, refining fault localization at the statement level reveals further details for identifying failure causes.

We ensure a feeling of immediacy when exploring behavior by splitting run-time analysis of test cases over multiple runs [18]. Usually, developers comprehend program behavior by starting with an initial overview of all run-time information and continuing with inspecting details. This systematic method guides our approach to dynamic analysis: run-time data is captured when needed. *Step-wise run-time analysis* consists of a first shallow analysis that represents an overview of a test run (a pure method call tree) and additional refinement analysis runs that record on-demand user-relevant details (e.g. state of variables, profiling data, statement coverage). Thereby, test cases fulfill the requirement to reproduce arbitrary points on a program execution in a short time [25]. Thus, by dividing dynamic analysis costs across multiple test runs, we ensure quick access to relevant run-time information without collecting needless data up front.

We classify behavior with respect to suspiciousness scores of methods for an efficient navigation to failure causes in large traces. Therefore, we either reuse PathMap's already ranked methods or rerun the spectrum-based fault localization on traced methods again. The trace is divided into more or less erroneous behavior depending on test results of called methods. On the analogy of PathMap, we colorize the trace with suspiciousness and confidence scores at each executed method. Moreover, a query mechanism supports the navigation to erroneous behavior. We expect that our classified traces identify failure causes more quickly as it allows shortcuts to methods that are likely to include the defect.

Analogous to step-wise run-time analysis, we are also able to refine fault localization at the statement level. For identifying failure causes in full detail, PathDebugger allows for refining spectrum-based fault localization inside specific methods. We run all covering tests, simulate byte code execution, and obtain required coverage information. We compute suspiciousness scores of statements with the same formulas as before. Combining spectrum-based fault localization and step-wise run-time analysis provides a good trade-off between performance and fault localization details. We restrict the performance decrease of statement-level analysis only to the method of interest and offer developers both fast access to erroneous behavior of methods and optionally refinements of suspicious statements.

In our Seaside example, PathDebugger highlights the erroneous behavior of creating buffered responses and supports developers in understanding how suspicious methods belong together. Following Figure 5, developers focus on the failing `testIsCommitted` behavior. They begin with the search for executed methods with a failure cause probability larger than 90 % (1). The trace includes and highlights four methods matching this query. Since the `writeContentOn:` method (2) has been executed shortly before the failure occurred, it should be favored for exploring corrupted state

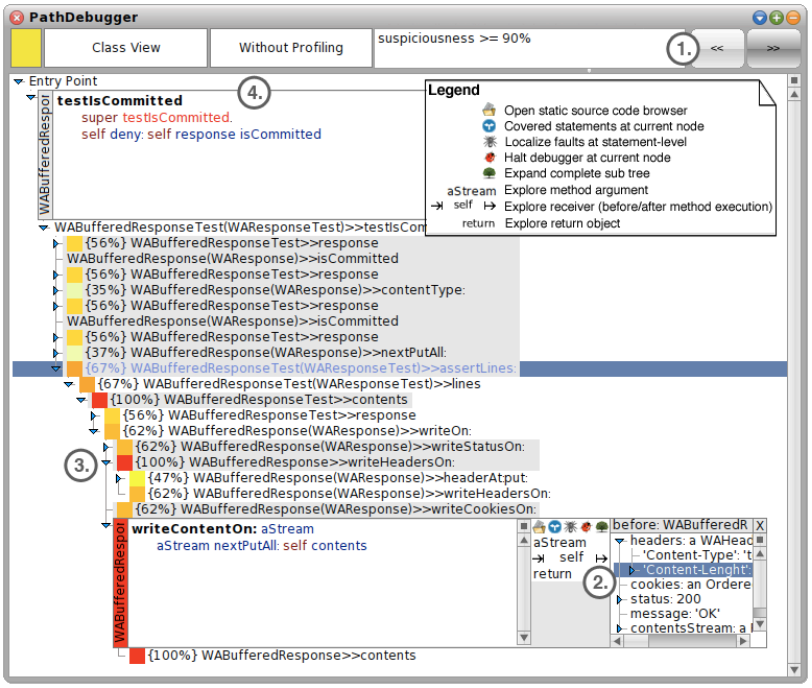

Fig. 5 PathDebugger is our lightweight back in time debugger that classifies failing test behavior for supporting developers in navigating to failure causes.

and behavior first[†4]. A detailed inspection of the receiver object reveals that the typo already exists before executing this method. Following the infection chain backwards, more than three methods can be neglected before the next suspicious method is found (3). Considering `writeHeadersOn:` in the same way manifests the failure cause. If necessary, developers are able to refine fault localization at the statement-level and see that only the first line of the test case is always executed, thus triggering the fault (4). Although PathDebugger supports developers in comprehending programs [18], it depends on experience if they recognize that all suspicious methods are part of the response creation.

3.4 Identifying Developers as Contact Persons

As understanding failure causes still requires thorough familiarity with suspicious system parts, we propose a new metric for identifying expert knowledge. 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 [2]. 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. 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.

First, from all methods of our system under observation ($M_{Partial}$) we create a new set that includes methods being identified by the spectrum-

†4 The simple accessor method `contents` can be neglected at this point.

based fault localization.

$$M_{Suspicious} = \{m \in M_{Partial} \mid suspicious(m) > 0\}$$

Second, with the help of Smalltalk's source code management system we identify developers that have implemented at least one of these suspicious methods. Having this list, we divide suspicious methods into one set per developer based on the method's last author.

$$M_{Developer} = \{m \in M_{Suspicious} \mid authorOf(m) = Developer\}$$

Third, for a specified set of methods we sum up suspiciousness and confidence scores and create a weighted average of both. The harmonic mean combines both values and prevents outliers such as high suspiciousness but low confidence.

$$FScore(M) = 2 \cdot \frac{\left(\sum_{m \in M} suspicious(m)\right) \cdot \left(\sum_{m \in M} confidence(m)\right)}{\sum_{m \in M} suspicious(m) + confidence(m)}$$

Fourth, we normalize individual developer scores by comparing them with the value of all suspicious methods.

$$developerRanking(Developer) = \frac{FScore(M_{Developer})}{FScore(M_{Suspicious})}$$

Finally, we sort all developers by their achieved expert knowledge so that we estimate the most qualified contact persons even though the cause is not yet known.

Table 1 Our developer ranking points out (anonymized) experts.

Developer	Ranking	Suspiciousness	Confidence	F-Measure
A	68 %	13.6	17.3	15.2
B	26 %	5.8	6.1	5.9
C	4 %	1.0	0.7	0.8
D	1 %	0.3	0.2	0.2

With respect to our typing error, we reduce the number of potential contact persons to 4 out of 24 Seaside developers, whereby the author of the failure-inducing method is marked as particularly important. Table 1 summarizes the (interim) results of our developer ranking metric and suggests Developer A for fixing the defect by a wide margin. With our fault-based team navigation, 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.

4 Evaluation

We evaluate test-driven fault navigation with respect to its practicability for developers, the accuracy of our developer ranking metric, and the efficiency of our Path tool suite[†5].

4.1 Practicability of Test-Driven Fault Navigation

For evaluating our process and tools, we conduct a user study within the scope of a real world project. We observe different developers while debugging several failures and compare our approach with standard debugging tools.

Experimental Setup

We choose the *Orca* Web framework as the underlying software system for our user study. Orca allows to implement Web applications in a single object-oriented language and was developed by eight students in nine months. They worked full time on the project and in close collaboration with an industrial partner. We restrict the study to Orca's core packages (Web server, Javascript translator, and object cache) whose properties are summarized in Table 5.

Since the required debugging effort depends on individual skills and knowledge about the system,

†5 Raw data at `http://www.hpi.uni-potsdam.de/swa/tmp/tdfnEvaluation.xlsx`

we determined by a questionnaire four developers that have similar skills but different understandings of Orca's core packages. All selected students have more than five years of programming experience and professional expertise with symbolic debuggers. Regarding comprehension, the first developer called *low* has implemented nothing in one of the core classes, the second developer called *mid* has last seen the code two months ago, and the remaining two developers called *high 1* and *high 2* have recently changed large parts of the core.

The students should localize four typical failures, which are described in Table 2. Failure A returns a specific session object instead of all sessions. Failure B breaks the parsing loop too soon. Failure C cannot cache a null object. Failure D forgets to select a collection and processes with false clients. Each failure is reproducible with 4-15 failing test cases.

We conduct the user study by observing our developers during debugging Orca's failures. First, we introduced test-driven fault navigation to the participants within 30 minutes followed by one hour of instructed practice with our tools. Second, we chose for each developer two failures for debugging with standard tools and two failures for our Path tools. Finally, we observed them during debugging, measured the required time, and interviewed them afterwards. If the defect has not been localized after 20 minutes, we marked the failure as not solved.

Lessons Learned

We evaluate the influence of test-driven fault navigation for each developer. Table 3 summarizes the required time for debugging with and without our tools.

Although the low developer has nearly no understanding of the system, he was able to solve both failures with our approach in about 12 minutes. In contrast, failure B could not be solved with standard tools. After 20 minutes we allowed him to apply our Path tools and he localized the defect in less than two minutes. As the last defect was on the stack of the symbolic debugger, he identified the cause straightforwardly.

Table 3 Comparison of debugging tools (column: developer, row: failure, TFN: Test-driven Fault Navigation, STD: Standard Tools, N/S: not solved, time in minutes).

	Low	Mid	High 1	High 2
A	12:10 (TFN)	11:51 (STD)	5:04 (STD)	4:04 (TFN)
B	N/S (STD)	11:53 (STD)	2:21 (TFN)	1:21 (TFN)
C	12:00 (TFN)	5:24 (TFN)	2:38 (STD)	1:20 (STD)
D	4:30 (STD)	4:38 (TFN)	2:21 (TFN)	1:45 (STD)

The mid developer requires twice as much time with standard tools as with our Path tools. He identified infected state still fast but following the infection chain back was cumbersome. While debugging failure B with standard tools he mentioned: *"I know the corrupted state but I cannot remember where does it come from. I need a trace of what happened before."* In the case of failure C and D, the classification of suspicious entities allowed him to invent better hypotheses about the failure cause and to abbreviate the execution history.

Due to their good program understanding both high developers invent proper diagnosis for the failure cause quite fast. Even if both tool chains have no significant differences in debugging time, both developers confirmed that the suspiciousness scores further strengthen their hypotheses. *High 1* stated: *"The coloring of map and trace has helped a lot in focusing on suspicious entities."* The short debugging time of failure B, C, and D came into being because there were only two suspicious methods, they jumped directly to the initialization method,

Table 2 Description of Orca's failures.

Failure	Description	Orca Package	Defect on stack	Localization rank
A	Wrong return value	Server	No	10 (1.0)
B	False loop condition	Translator	No	2 (0.76)
C	Not initialized object	Cache	No	17 (1.0)
D	Missing selection	Server	Yes	8 (1.0)

or the defect was apparently on the debugger stack. Nevertheless, one of them concluded *"I can very well imagine that the Path tools improve debugging of our real failures, too."*

During test-driven fault navigation, all developers take advantage of the combined perspectives of our Path tools. In doing so, they usually started with a breadth-first search and used PathMap for the first two minutes. The rest of the time, they chose an erroneous test case and followed the infection chain through suspicious behavior. Only in the case of failure B, no run-time information was required for developer *high 2*.

With the help of our user study, we conclude that test-driven fault navigation is able to decrease debugging costs with respect to required time and developer's effort. Especially, developers with less system knowledge, which is often the case in maintaining legacy systems, profit from reduced debugging time. Also all developers confirmed that our approach is promising in debugging with less mental effort and that our tools enable a feeling of immediacy [18]. It appeared easier for them to create proper hypotheses about failure causes without to be slowed down by our approach.

4. 2 Accuracy of Recommended Developers

We evaluate our developer ranking metric by introducing a considerable number of defects into the Seaside Web framework. We randomly chose 1,000 covered methods that are neither part of test code nor trivial (e. g., getters). For each method, we insert a defect (hiding the method body and returning the receiver object), compute a sorted list of recommended developers, and compare it with the last author of the faulty method. We assume this person is the most qualified developer for explaining the failure cause.

Table 4 Average developer ranking results

Developer rating	Developer rank	Number of developers
0.439	1.75	7

Table 4 presents the average scores of our developer ranking metric. We identify the most qualified developers with a rating of 43 % and between the first two positions out of seven recommended developers. Figure 6 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.

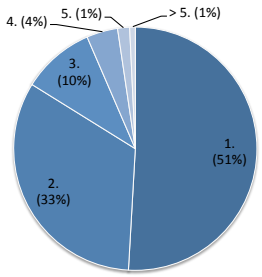

Fig. 6 Rankings of the most qualified developers.

4.3 Efficiency of the Path Tool Suite

We evaluate the overhead of our Path tools by measuring the total time for collecting and presenting run-time information from four different Smalltalk projects. The project properties are summarized in the upper part of Table 5. The test cases cover system, acceptance, and unit tests, imposing different computational costs. Of the four projects, two (Orca and AweSOM) are research prototypes developed in our group. The remaining two are production-quality projects and in daily use in software development and business activities. For measuring the run-time overhead PathMap executes the entire unit test suite with and without fault localization, PathDebugger runs each test on its own and analyzes the overhead produced by step-wise run-time analysis. All experiments were run on a MacBook with a 2.4 GHz Intel Core 2 Duo and 4 GB RAM running Mac OS X 10.6.6, using Squeak version 4.1 on a 4.2.1b1 virtual machine.

The average results for each project are described in the lower part of Table 5. The first two rows show the time (in seconds) required for executing all tests and the overhead resulting from spectrum-based fault localization. PathMap's fault localization slows down execution by a factor of 1.7 to 3.9. The variation originates from additional instrumentation and visualization costs. Nevertheless, this overhead is low enough for applying spectrum-based fault localization frequently.

The last four rows list the average run-time per test and the overhead associated with building the lightweight call tree, reloading state, and refining fault localization at statements. For all tests and projects, each kind of dynamic analysis is done in less than 500 ms on average. The 99th percentile for the shallow analysis overhead is below 750 ms. Incremental refinement imposes a minimum overhead in most cases: the 95th percentile is below 25 ms for all tests. The same is true for refining fault localization with the 95th percentile below 20 ms. This supports fast response times when debugging a test execution back in time since run-time data is provided in considerably less than two seconds [18].

4.4 Threats to Validity

The Smalltalk context of our evaluation might impede validity by limited scalability and general applicability. However, the Seaside Web framework is in fact a real-world system and it exhibits source code characteristics comparable to particular complex Java systems such as JHotDraw [18]. Even if the remaining projects were developed in parts by ourselves, they illustrate the applicability of our Path tools once unit tests are available. While these insights do not guarantee scalability to arbitrary languages and systems, they provide a worthwhile direction for future studies assessing general applicability.

The evaluation setting has four characteristics that might limit validity. First, our user study is only based on a small student project. Although we require a larger study for a general conclusion, our user study still reveals the benefits of our process and tool suite. Moreover, we consider our students and their Orca project as a real world scenario. Each developer has more than five years of programming experience and they worked full time

Table 5 Project characteristics and average run-time analysis

	Orca	Seaside	Compiler	AweSOM
Classes	60	394	64	68
Methods	848	3708	1294	742
Tests	68	674	49	124
Coverage	68.6 %	58.3 %	51.1 %	81.8 %
Execution time all tests (s)	7.17	9.19	0.91	3.77
Δ Fault Localization (s)	4.77	26.61	1.85	8.70
Execution time per test (ms)	74.98	0.76	7.69	17.33
Δ Shallow Analysis (ms)	389.50	336.17	247.23	235.79
Δ Refinement Analysis (ms)	10.27	16.92	2.15	5.93
Δ Refined Fault Localization (ms)	15.81	1.23	9.76	16.29

on Orca for nine months. Second, the accuracy of our developer ranking metric might be limited by the fact that after a fundamental refactoring about 90 % of Seaside's methods belong to three main authors. For this reason, we have always a probability of circa 30 % to guess the best developer. Nevertheless, the accuracy of our metric suggest for every second failure the right contact person. Third, garbage collection was disabled during measurement to elide performance influences. In a realistic setting with enabled garbage collection, minimal slowdowns would be possible. Finally, we rely on tests to obey certain rules of good style: e. g., they should be deterministic. Tests that do not follow these guidelines might hamper our conclusions. The tests that we used in our evaluation were all acceptable in this respect.

5 Related Work

We divide related work into three categories corresponding to our PathMap, PathDebugger, and developer ranking metrics: spectrum-based fault localization, back in time debugging, and approaches to determine developer expertise.

5.1 Spectrum-based Fault Localization

Spectrum-based fault localization is an active field of research where passing and failing program runs are compared with each other to isolate suspicious behavior or state. Tarantula [9] analyzes and visualizes the overlapping behavior of test cases with respect to their results. At the system overview level, each statement is represented as a line of pixels and colorized with a suspiciousness score that refers to the probability of containing the defect. Later, Gammatella [17] presents a more scalable and generalized visualization in form of a tree map but only for classes. The Whither tool [22] collects spectra of several program executions, determines with the nearest neighbor criterion the most similar correct and faulty run, and creates a list of suspicious differences. AskIgor [5] identifies state differences of passed and failed test runs and automatically isolates the infection chain with delta debugging and cause transitions. A first empirical study [8] comparing these different spectrum-based approaches concludes that the Tarantula technique is more effective and efficient than the other ones. A more comprehensive study [1], investigating the impact of metrics and test design on the diagnostic accuracy of fault localization, states that similarity metrics are largely independent of test design and

that the Ochiai coefficient consistently outperforms all other approaches.

All presented approaches produce ranked source code entities that are likely to include failure causes. However, as defects are rarely localized without doubt, developers have to determine the remaining results by hand. We argue that our presented test-driven fault navigation deals with this issue. It combines multiple perspectives based on already gathered suspiciousness information and supports developers in further approximating the real failure cause.

5.2 Back in Time Debugging

To follow the infection chain from the observable failure back to its cause, back-in time debuggers allow developers to navigate an entire program execution and answer questions about the cause of a particular state. The omniscient debugger [12] records every event, object, and state change until execution is interrupted. However, the required dynamic analysis is quite time- and memory-consuming. Unstuck [7] is the first back-in time debugger for Smalltalk but suffers from similar performance problems. WhyLine [11] allows developers to ask a set of *"why did"* and *"why didn't"* questions such as why a line of code was not reached. However, WhyLine requires a statically-typed language and it does not scale well with long traces. Other approaches aim to circumvent these issues by focusing on performance improvements in return for a more complicated setup. The trace-oriented debugger [21] combines an efficient instrumentation for capturing exhaustive traces and a specialized distributed database. Later, a novel indexing and querying technique [20] ensures scalability to arbitrarily large execution traces and offers an interactive debugging experience. Object flow analysis [13] in conjunction with object aliases also allows for a practical back in time debugger. The approach leverages the virtual machine and its garbage collector to remove no longer reachable objects and to discard corresponding events.

Compared to such tools, our PathDebugger is a lightweight and specialized back in time debugger for localizing failure causes in unit tests. Due to step-wise run-time analysis, we do not record each event beforehand but rather split dynamic analysis over multiple runs. Furthermore, our classified traces allow to hop into erroneous behavior directly. Without this concept, developers require more internal knowledge to isolate the infection chain and to decide which path to follow.

5.3 Determining Developer Expertise

Our developer ranking metric is mostly related to approaches that identify expert knowledge for development tasks. The expertise browser [16] quantifies people with desired knowledge by analyzing information from change management systems. XFinder [10] is an Eclipse extension that recommends a ranked list of developers to assist with changing a given file. A developer-code map created from version control information presents commit contributions, recent activities, and the number of active workdays per developer and file. The Emergent Expertise Locator [15] approximates, depending on currently opened files and their histories, a ranked list of suitable team members. An empirical study [6] verifies the assumption that programmer's activity indicates some knowledge of code and presents additional factors that also indicate expertise knowledge such as authorship or performed tasks. Besides common expertise knowledge, there are other approaches that focus on assigning bug reports to the most qualified developers. A first semi-automated machine learning approach [2] works on open bug repositories and learns from already resolved reports the relationship between developers and bugs. It classifies new

incoming reports and recommends a few developers that have worked on similar problems before. Develect [14] applies a similar approach but it matches the lexical similarities between the vocabulary of bug reports and the diffs of developers' source code contributions.

In contrast to our developer ranking metric, previous approaches are generally applicable but their recommendation accuracy is limited. Our metric is specialized for debugging and recommends in most cases a suitable contact person. Although we require at least one failing test case, we think that often its implementation can be derived from bug reports.

6 Conclusion

We propose *test-driven fault navigation* as a process and accompanying tool suite for debugging failures reproducible via unit tests. A systematic breadth-first search guides developers to failure causes within structure and behavior. Corresponding expert members of the development team are ranked as potential candidates for fixing these problems. With the help of *PathMap* and *PathDebugger*, developers can localize suspicious system parts, debug erroneous behavior back to failure-inducing origins, and learn about other developers who are likely able to help. Our evaluation and case study demonstrate that the combination of unit tests, spectrum-based fault localization, and *test-driven fault navigation* is practical for bringing developers closer to failure causes.

Future work is two-fold. Our approach will be extended to take version control information into account such that the change history of methods further reduces search space and to better propose more suitable experts. Also, we are planning a larger user study to assess how test-driven fault navigation improves more general debugging activities.

References

[1] Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J.: A practical evaluation of spectrum-based fault localization, *JOSS*, Vol. 82, No. 11(2009), pp. 1780–1792.

[2] Anvik, J., Hiew, L., and Murphy, G. C.: Who Should Fix this Bug?, *ICSE*, 2006, pp. 361–370.

[3] Beck, K.: *Test-driven Development: By Example*, Addison-Wesley Professional, 2003.

[4] Brant, J., Foote, B., Johnson, R., and Roberts, D.: Wrappers to the Rescue, *ECOOP*, 1998, pp. 396–417.

[5] Cleve, H. and Zeller, A.: Locating Causes of Program Failures, *ICSE*, 2005, pp. 342–351.

[6] Fritz, T., Murphy, G. C., and Hill, E.: Does a Programmer's Activity Indicate Knowledge of Code?, *ESEC-FSE*, 2007, pp. 341–350.

[7] Hofer, C., Denker, M., and Ducasse, S.: Design and Implementation of a Backward-in-Time Debugger, *NODe*, 2006, pp. 17–32.

[8] Jones, J. A. and Harrold, M. J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, *ASE*, 2005, pp. 273–282.

[9] Jones, J. A., Harrold, M. J., and Stasko, J.: Visualization of Test Information to Assist Fault Localization, *ICSE*, 2002, pp. 467–477.

[10] Kagdi, H., Hammad, M., and Maletic, J.: Who Can Help Me with this Source Code Change?, *ICSM*, 2008, pp. 157–166.

[11] Ko, A. J. and Myers, B. A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *ICSE*, 2008, pp. 301–310.

[12] Lewis, B.: Debugging Backwards in Time, *AADEBUG*, 2003, pp. 225–235.

[13] Lienhard, A., Gîrba, T., and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, *ECOOP*, 2008, pp. 592–615.

[14] Matter, D., Kuhn, A., and Nierstrasz, O.: Assigning Bug Reports Using a Vocabulary-based Expertise Model of Developers, *MSR*, 2009, pp. 131–140.

[15] Minto, S. and Murphy, G. C.: Recommending Emergent Teams, *MSR*, 2007, pp. 5–14.

[16] Mockus, A. and Herbsleb, J. D.: Expertise Browser: A Quantitative Approach to Identifying Expertise, *ICSE*, 2002, pp. 503–512.

[17] Orso, A., Jones, J., and Harrold, M. J.: Visualization of Program-Execution Data for Deployed Software, *SoftVis*, 2003, pp. 67–76.

[18] Perscheid, M., Steinert, B., Hirschfeld, R., Geller, F., and Haupt, M.: Immediacy through Interactivity: Online Analysis of Run-time Behavior, *WCRE*, 2010, pp. 77–86.

[19] Perscheid, M., Tibbe, D., Beck, M., Berger, S., Osburg, P., Eastman, J., Haupt, M., and Hirschfeld, R.: *An Introduction to Seaside*, Software Architecture Group (Hasso-Plattner-Institut), 2008.

[20] Pothier, G. and Tanter, E.: Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging, *ECOOP*, 2011, pp. to appear.

[21] Pothier, G., Tanter, E., and Piquer, J.: Scalable Omniscient Debugging, *OOPSLA*, 2007, pp. 535–552.

[22] Renieres, M. and Reiss, S.: Fault Localization with Nearest Neighbor Queries, *ASE*, 2003, pp. 30–39.

[23] Richardson, D. J. and Thompson, M. C.: An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection, *IEEE TSE*, Vol. 19(1993), pp. 533–553.

[24] Shneiderman, B.: Tree Visualization with Tree-Maps: 2-D Space-Filling Approach, *ACM Trans. Graph.*, Vol. 11, No. 1(1992), pp. 92–99.

[25] Steinert, B., Perscheid, M., Beck, M., Lincke, J., and Hirschfeld, R.: Debugging into Examples: Leveraging Tests for Program Comprehension, *TestCom*, 2009.

[26] Vessey, I.: Expertise in Debugging Computer Programs: A Process Analysis, *Int. J. Man Mach. Stud.*, Vol. 23, No. 5(1985), pp. 459–494.

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