Demonstrating
Test-driven Fault Navigation

Michael Perscheid

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

In this paper, we present a demonstration of our test-driven fault navigation approach. This interconnected guide for debugging reproducible failures analyzes failure-reproducing test cases and supports developers in following infection chains back to root causes. With the help of a motivating error from the Seaside Web framework, we show how to reveal suspicious system parts, identify experienced developers for help, and debug erroneous behavior and state in the execution history.

1 Introduction

The correction of ubiquitous software failures can cost a lot of money [19] because their debugging is often a time-consuming development activity [3, 7]. During debugging, developers largely attempt to understand what causes failures: Starting with a test case, which reproduces the observable failure, they have to follow failure causes on the infection chain back to the root cause (defect) [22]. This idealized procedure requires deep knowledge of the system and its behavior because failures and defects can be far apart [8]. Unfortunately, common debugging tools are inappropriate to systematically investigate such infection chains in detail [9]. Thus, developers have to primarily rely on their intuition and the localization of failure causes takes up a lot of time [10]. To prevent debugging by trial and error, experienced developers apply the scientific method and its systematic hypothesis-testing [10,22]. However, even when using the scientific method the search for failure causes can still be a laborious task. First, missing expertise about the system makes it hard to understand incorrect behavior and to create reasonable hypotheses [11]. Second, contemporary debugging approaches still provide little or no support for the scientific method. For these reasons, we summarize our research question as follows:

> How can we effectively support developers in creating, evaluating, and refining failure cause hypotheses so that we reduce debugging costs with respect to time and effort?

In this paper, we present a demonstration of our *test-driven fault navigation* that guides the debugging of reproducible failures [13, 14, 20]. Based on the analysis of passing and failing test cases, we reveal anomalies and integrate them into a breadth first search that leads developers to defects. This systematic search consists of four specific navigation techniques that together support the creation, evaluation, and refinement of failure cause hypotheses for the scientific method. First, structure navigation [14] localizes suspicious system parts and restricts the initial search space. Second, team navigation [14] recommends experienced developers for helping with failures even if defects are still unknown. Third, behavior navigation [14, 15] allows developers to follow emphasized infection chains of failing test cases backwards. Fourth, state navigation [5, 6] identifies corrupted state and reveals parts of the infection chain automatically. We implement test-driven fault navigation in our *Path tools framework* [12, 14, 15, 20] for the Squeak/Smalltalk development environment and limit its computation costs with the help of our *incremental dynamic analysis* [5, 12, 15]. This lightweight dynamic analysis ensures a feeling of immediacy when debugging with our tools by splitting the run-time overhead over multiple test runs depending on developers' needs. Hence, our test-driven fault navigation in combination with our incremental dynamic analysis answers important questions in a short time: where to start debugging, who understands failure causes best, what happened before failures, and which program entities are in question.

The remainder of this paper is organized as follows: Section 2 explains the motivating example for demonstrating our approach. Section 3 briefly introduces our test-driven fault navigation. Section 4 demonstrates our approach and how it supports debugging of the motivating example. Section 5 concludes and presents next steps.

2 Motivating Example: Typing Error in Seaside

We introduce a motivating example error taken from the Seaside Web framework [2,16] that serves as a basis for demonstrating our test-driven fault navigation approach in the following sections.

Seaside[1] is an open source Web framework implemented in Smalltalk [4]. The framework provides a uniform, pure object-oriented view of Web applications and combines a component-based with a continuation-based approach [17]. With this, every component has its own control flow which leads to high reusability, maintainability and a high level of abstraction. Additionally, it is written in Smalltalk that allows developers to debug and update applications on the fly. It provides a layer over HTTP and HTML that let you build highly interactive Web applications that come very close to the implementation of real desktop applications. Finally, Seaside consists of about 650 classes, 5,500 methods and a large test suite with more than 700 test cases.

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. Once a client opens a Seaside Web application, its Web browser sends a request to the corresponding Web server. This request is then processed by the framework leading to a corresponding response to the browser. Depending on the Web application, this response is either a streamed or buffed response object. While the first transfers the

[1]`www.seaside.st`

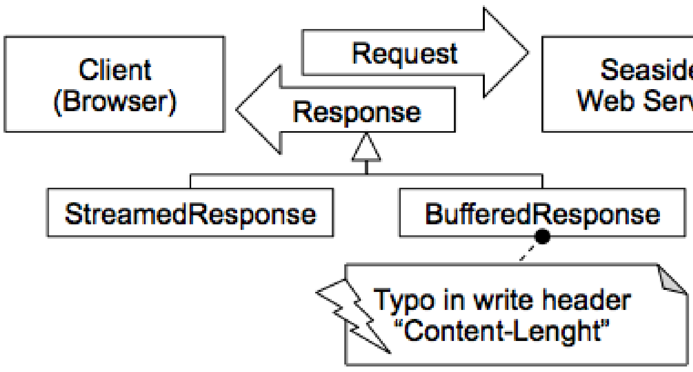


Figure 1: An inconspicuous typo in writing buffered response headers leads to faulty results of several client requests.

message body as a stream, the latter buffers and sends the response as a whole. During the creation of buffered responses, there is a typo in writing the header. The typo in "Content-Leng<u>ht</u>" is inconspicuous but leads to invalid results in browser 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 typos [18].

3 Anomalous Guide to Localize Causes in Failing Test Cases

Based on test cases that reproduce the observable failure [20], we introduce a novel systematic top-down debugging process with corresponding tools called test-driven fault navigation. It does not only support the scientific method with a breadth-first search [21] but also integrates hidden test knowledge for guiding developers to failure causes. Starting with a failure-reproducing test case as entry point, we reveal suspicious system parts, identify experienced developers for help, and navigate developers along the infection chain step by step. In doing so, anomalies highlight corrupted behavior and state and so assist developers in their systematically hypothesis-testing. Figure 2 summarizes our complete test-driven fault navigation process and its primary activities:

Reproducing failure: As a precondition for all following activities, developers have to reproduce the observable failure in the form of at least one test case. Besides the beneficial verification of resolved failures, we require tests above all as entry points for analyzing erroneous behavior. For this activity, 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 [1].

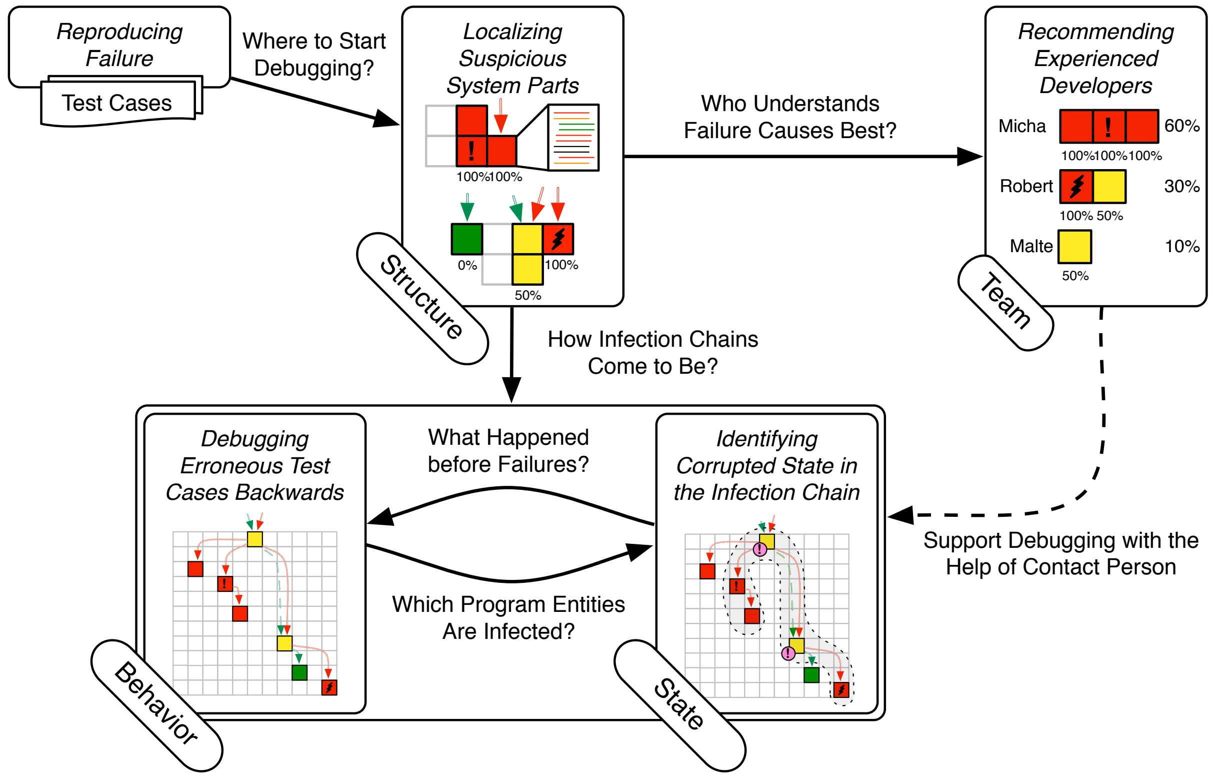

Reproducing Failure
Test Cases
Where to Start Debugging?
Localizing Suspicious System Parts
100%100%
0%
100%
50%
Structure
Who Understands Failure Causes Best?
Recommending Experienced Developers
Micha
60%
100%100%100%
Robert
30%
100% 50%
Malte
10%
50%
Team
How Infection Chains Come to Be?
Debugging Erroneous Test Cases Backwards
What Happened before Failures?
Which Program Entities Are Infected?
Identifying Corrupted State in the Infection Chain
Behavior
State
Support Debugging with the Help of Contact Person

Figure 2: Our test-driven fault navigation debugging process guides developers with interconnected advice to reproducible failure causes in structure, team, behavior, and state of the system under observation.

Localizing suspicious system parts (*Structure navigation*) Having at least one failing test, developers can compare its execution with other test cases and identify structural problem areas that help in creating initial hypotheses. 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 supports spectrum-based fault localization within the system structure. It provides a scalable tree map visualization and a low overhead analysis framework that computes anomalies at methods and refines results at statements on demand.

Recommending experienced developers (*Team navigation*) Some failures require knowledge of experts to help developers in creating proper hypotheses. 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 even if the defect is still unknown. 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.

Debugging erroneous test cases backwards (*Behavior navigation*) For refining their understanding of erroneous behavior, developers experiment with the execution and state history of a failing test case. To follow the infection chain back to the defect, they choose a proper entry point such as the failing test or one of its suspicious methods and start *PathFinder* our lightweight back in time debugger. If anomalies are available, we classify the executed trace and so allow developers to create proper hypotheses that assist the behavioral navigation to defects.

Identifying corrupted state in the infection chain (*State navigation*) Besides the classification of executed behavior with spectrum-based anomalies, we also highlight parts of the infection chain with the help of state anomalies. We derive state properties from the hidden knowledge of passing test cases, create generalized contracts, and compare them with failing tests. Such dynamic invariants reveal state anomalies by directly violating contracts on the executed infection chain and so assist developers in creating and refining hypotheses. For this state navigation, our *PathMap* automatically harvests objects and creates contracts while our *PathFinder* integrates the violations into the execution history.

Besides our systematic top down process for debugging reproducible failures, the combination of testing and anomalies also provides the foundation for interconnected navigation with a high degree of automation. All four navigation activities and their anomalous results are affiliated with each other and so allow developers to explore failure causes from combined perspectives. An integration supports developers in answering more difficult questions and allows other debugging tasks to benefit even from anomalies. Linked views between suspicious source code entities, erroneous behavior, and corrupted state help not only to localize causes more efficiently but also to identify the most qualified developers for understanding the current failure. Our *Path tools* support these points of view in a practical and scalable manner with the help of our incremental dynamic analysis. With a few user interactions, we split the expensive costs of dynamic analysis over multiple test runs and varying granularity levels so that we can provide both short response times and suitable results. Thus, developers are able to answer with less effort where to start debugging; who understands failure causes best; what happened before failures; and which program entities are infected.

4 Example: Debugging Seaside's Typo

With respect to our debugging process, we demonstrate each navigation step with the help of Seaside's typing error in more detail. Therefore, we start with the implementation of a failing test case that reproduce the observable failure. In the case of our example, developers have to implement a simple server request waiting for a corrupted response that cannot be parsed correctly. After that, they are able to apply our approach and its specific navigations as follows:

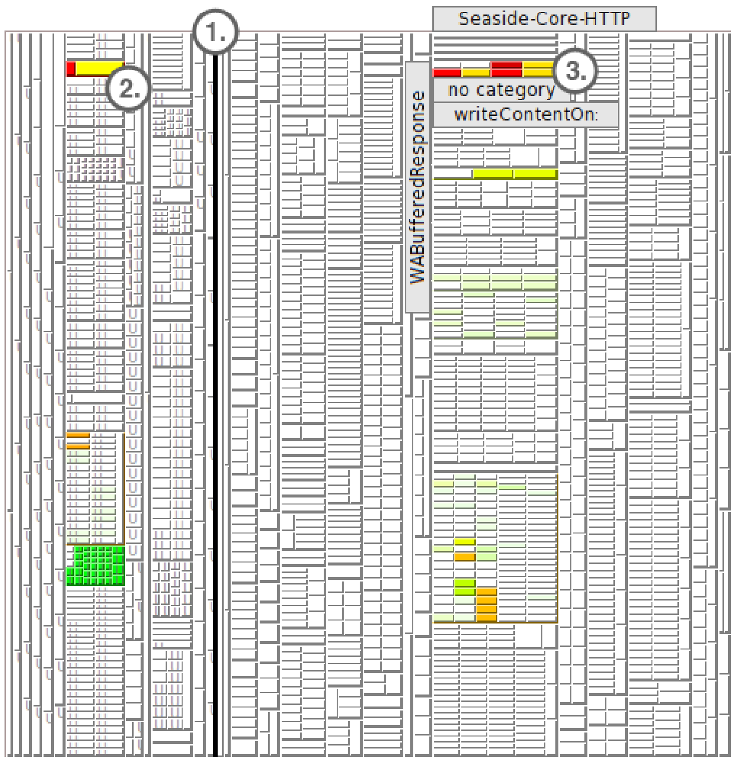
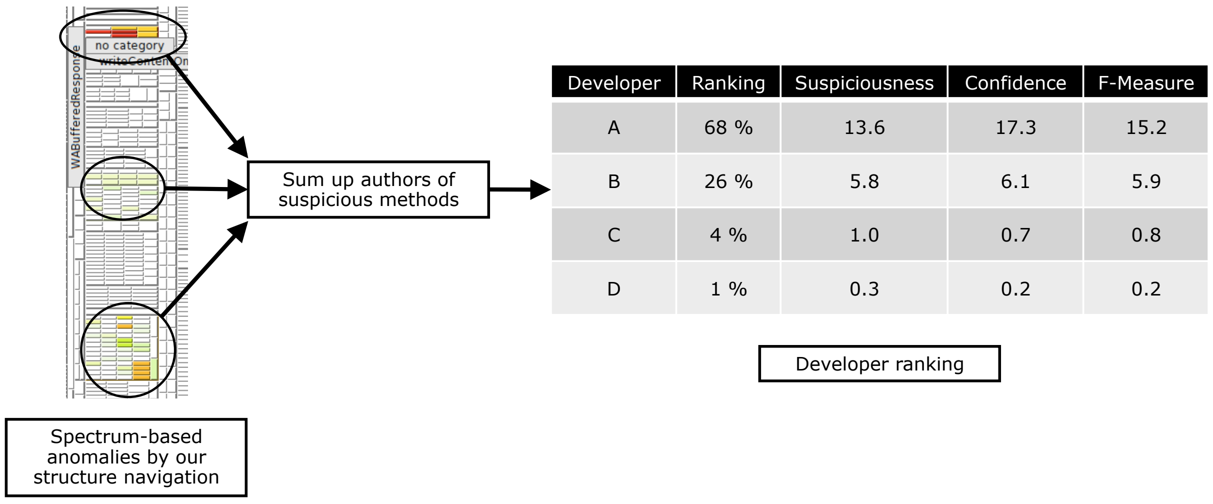


Figure 3: PathMap: In our Seaside example, our structure navigation restricts the search space to a few very suspicious methods in buffered responses.

4.1 Structure Navigation: Localizing Suspicious Response Objects

In our motivating typing error, we localize several anomalies within Seaside's response methods. Figure 3 presents the tree map visualization of Seaside with test classes on the left side and application classes on the right side (1). After running Seaside's response test suite with the result of 53 passed and 9 failed tests, our structure navigation colorizes the suspiciousness scores of methods and reveals anomalous areas of the system. For example, the interactively explorable yellow box (2) illustrates that all nine failing tests are part of the buffered test suite. In contrast, the green box below includes the passed streaming tests. The more important information for localizing failure causes is visualized at the upper right corner (3). There are three red and three 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:` and `content` method shows that they shares the same characteristics as our failure cause in `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 further help in order to understand how the failure comes to be.


no category
writeContentOn
WABufferedResponse
Sum up authors of suspicious methods
Developer ranking
Spectrum-based anomalies by our structure navigation

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

Figure 4: PathMap: Our developer ranking points out (anonymized) experts. Based on authors of spectrum-based anomalies, we create a ranked list of possible experts that understand failure causes best.

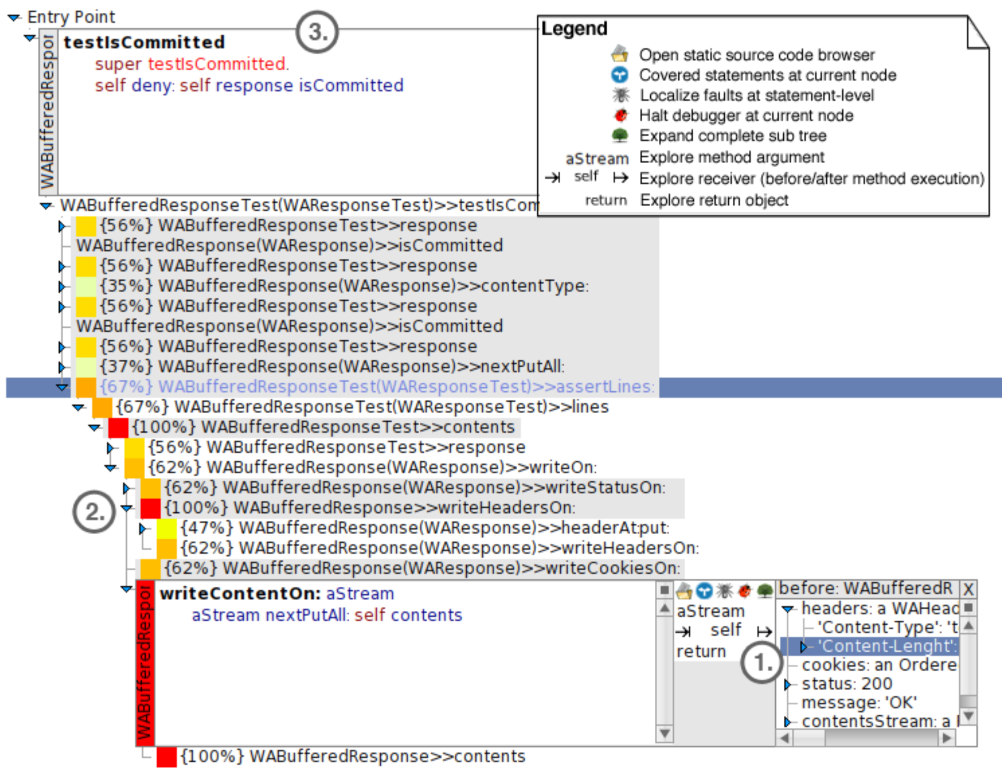
4.2 Team Navigation: Finding Experienced Seaside Developers for Help

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. The table in Figure 4 summarizes the (interim) results of our developer ranking metric and suggests Developer A[2] for fixing the defect by a wide margin. Compared to a coverage-based metric, which simply sums up covered methods of failing tests per developer, our results are more precise with respect to debugging. A's lead is shrinking (only 55 %), C (24 %) changes the place with B (19 %), and the list is extended with a fifth developer (1 %). It should be noted that our team navigation does not blame developers. We expect that the individual skills of experts help in comprehending and fixing failure causes more easily and thus might reduce the overall costs of debugging.

4.3 Behavior Navigation: Understanding How the Failure Comes to Be

In our Seaside example, we highlight the erroneous execution history of creating buffered responses and support developers in understanding how suspicious methods belong together. Following Figure 5, developers focus on the failing `testIsCommitted` behavior and follow the shortest infection chain from the observable failure back to its root cause. They begin with the search for executed methods with a failure cause probability larger than 90 %. The trace includes and highlights four methods matching this query. Since the `writeContentOn:` method (1) has been executed shortly before the failure occurred, it should be favored for exploring corrupted

[2]Developers' names have been anonymized.



```
Entry Point
  testIsCommitted
      super testIsCommitted.
      self deny: self response isCommitted
  WABufferedResponseTest(WAResponseTest)>>testIsCom
    {56%} WABufferedResponseTest>>response
    WABufferedResponse(WAResponse)>>isCommitted
    {56%} WABufferedResponseTest>>response
    {35%} WABufferedResponse(WAResponse)>>contentType:
    {56%} WABufferedResponseTest>>response
    WABufferedResponse(WAResponse)>>isCommitted
    {56%} WABufferedResponseTest>>response
    {37%} WABufferedResponse(WAResponse)>>nextPutAll:
    {67%} WABufferedResponseTest(WAResponseTest)>>assertLines:
      {67%} WABufferedResponseTest(WAResponseTest)>>lines
        {100%} WABufferedResponseTest>>contents
          {56%} WABufferedResponseTest>>response
          {62%} WABufferedResponse(WAResponse)>>writeOn:
            {62%} WABufferedResponse(WAResponse)>>writeStatusOn:
            {100%} WABufferedResponse>>writeHeadersOn:
              {47%} WABufferedResponse(WAResponse)>>headerAt:put:
              {62%} WABufferedResponse(WAResponse)>>writeHeadersOn:
            {62%} WABufferedResponse(WAResponse)>>writeCookiesOn:
            writeContentOn: aStream
                aStream nextPutAll: self contents
            {100%} WABufferedResponse>>contents
```

Legend	
🏠	Open static source code browser
●	Covered statements at current node
✱	Localize faults at statement-level
●	Halt debugger at current node
●	Expand complete sub tree
aStream	Explore method argument
⇥ self ↦	Explore receiver (before/after method execution)
return	Explore return object

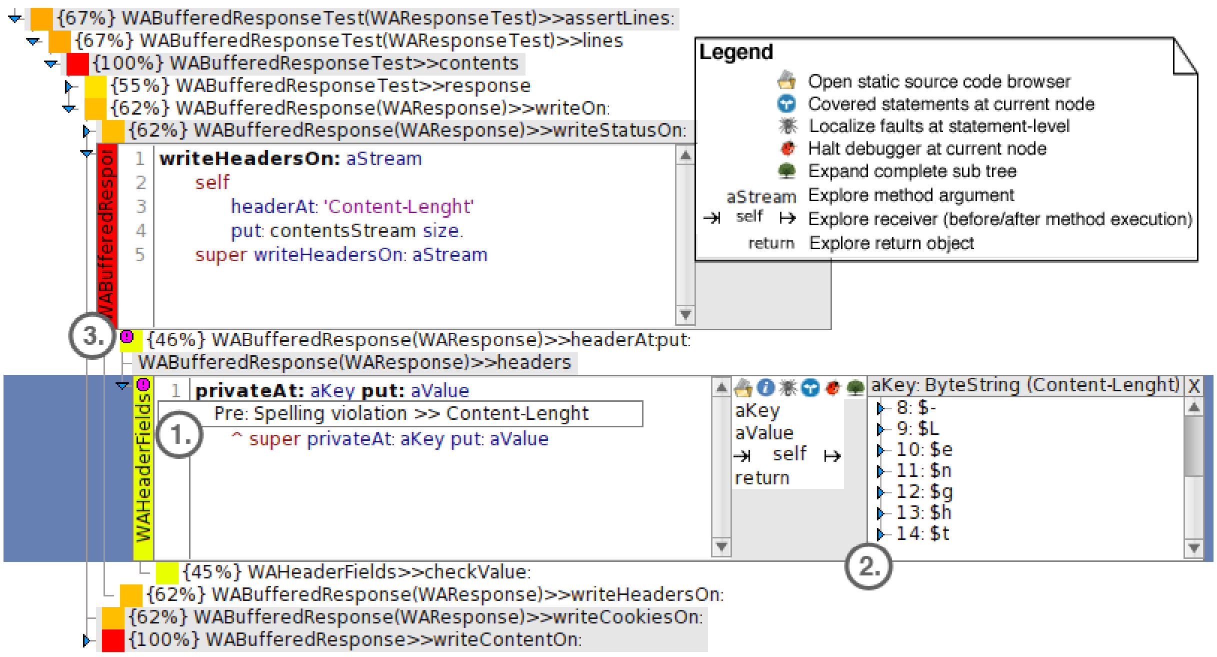
Figure 5: PathFinder: The classified execution history of our Seaside typing error.

state and behavior first[3]. 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 (2). Considering `writeHeadersOn:` in the same way manifests the failure cause. If necessary, developers are able to refine fault localization at the statement-level analogous to our structure navigation and see that only the first line of the test case is always executed, thus triggering the fault (3).

4.4 State Navigation: Come Closer to the Typing Error

In our Seaside typing error, our state navigation is able to reveal two anomalies close by the root cause. First, we run all passing tests from the still working streamed responses and collect type and value ranges of their applied objects. Among others we check all string objects if they are spelled correctly or not. Second, we derive common invariants from the concrete objects and create corresponding contracts. Thus, we propagate the implicit assertions of the response tests to each covered method and automatically generate assertions for pre-/post-conditions and invariants of the corresponding class. Each assertion summarizes common object properties such as types, value ranges of numbers, and permissions of undefined objects. Third, we execute the same failing test case as in our behavior navigation but now with enabled contracts. As soon as a

[3]The simple accessor method `contents` can be neglected at this point.



```
{67%} WABufferedResponseTest(WAResponseTest)>>assertLines:
  {67%} WABufferedResponseTest(WAResponseTest)>>lines
    {100%} WABufferedResponseTest>>contents
      {55%} WABufferedResponseTest>>response
      {62%} WABufferedResponse(WAResponse)>>writeOn:
        {62%} WABufferedResponse(WAResponse)>>writeStatusOn:
        WABufferedRespo
          writeHeadersOn: aStream
              self
                  headerAt: 'Content-Lenght'
                  put: contentsStream size.
              super writeHeadersOn: aStream
          {46%} WABufferedResponse(WAResponse)>>headerAt:put:
          WABufferedResponse(WAResponse)>>headers
          WAHeaderFields
            privateAt: aKey put: aValue
              Pre: Spelling violation >> Content-Lenght
              ^ super privateAt: aKey put: aValue
            {45%} WAHeaderFields>>checkValue:
          {62%} WABufferedResponse(WAResponse)>>writeHeadersOn:
        {62%} WABufferedResponse(WAResponse)>>writeCookiesOn:
        {100%} WABufferedResponse>>writeContentOn:
```

```
aKey
aValue
self
return
```

```
aKey: ByteString (Content-Lenght)
8: $-
9: $L
10: $e
11: $n
12: $g
13: $h
14: $t
```

Legend

	Open static source code browser
	Covered statements at current node
	Localize faults at statement-level
	Halt debugger at current node
	Expand complete sub tree
aStream	Explore method argument
⇥ self ↦	Explore receiver (before/after method execution)
return	Explore return object

Figure 6: PathFinder: State anomalies highlight the typing error and reveal the infection chain in the near of the defect.

contract is violated, we mark the corresponding exception in the execution history and so reveal for our `testIsCommitted` two state anomalies that are close by the defect.

Figure 6 summarizes the result of our state navigation. We mark method calls triggering a violation with small purple exclamation marks (1). Developers can further inspect these violations and see that a precondition fails. There is a spelling violation in the first argument of this method—all streamed responses used correctly spelled identifier keys for their header information. The corrupted state is opened for further exploration on the right (2). As our typo in “content-lenght” is automatically revealed, our state navigation gives developers helpful advice about the real failure cause. Another spelling violation is close by and developers can easily follow the infection chain back (3). Finally, the next very suspicious spectrum-based anomaly at `writeHeadersOn:` highlights the last step to the root cause. Following both state and spectrum-based anomalies directly guides developers to the defect of our Seaside typing error and also allows them to understand what causes the failure.

5 Summary and Next Steps

In this paper, we presented a demonstration of our test-driven fault navigation by debugging a small example. Starting with the structure navigation, we restrict the initial search space and lower speculations about failure causes. Based on this information, we are able to recommend experienced developers that can further help with debugging this failure. After that, developers apply our behavior and state navigation and follow the highlighted infection chain back to its root cause.

Future work deals with finishing the dissertation. So far, there is a first complete draft including 180 pages in total with 135 pages of content. We plan to submit the thesis at the end of this year.

References

[1] K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 1st edition, 2003.

[2] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A Flexible Environment for Building Dynamic Web Applications. *IEEE Software*, 24(5):56–63, 2007.

[3] M. Eisenstadt. My Hairiest Bug War Stories. *Communications of the ACM*, 40(4):30–37, 1997.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1st edition, 1983.

[5] M. Haupt, M. Perscheid, and R. Hirschfeld. Type Harvesting A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 1282–1289. ACM, 2011.

[6] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic Contract Layers. In *Proceedings of the Symposium on Applied Computing*, SAC, pages 2169–2175. ACM, 2010.

[7] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, ASID, pages 25–33. ACM, 2006.

[8] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 15–26. ACM, 2005.

[9] H. Lieberman and C. Fry. Will Software Ever Work? *Communications of the ACM*, 44(3):122–124, 2001.

[10] R. Metzger. *Debugging by Thinking - A Multidisciplinary approach*. Elsevier Digital Press, 1st edition, 2003.

[11] N. Palix, J. Lawall, G. Thomas, and G. Muller. How Often Do Experts Make Mistakes? In *Proceedings of the Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS, pages 9–15. Technical report 2010-33 Hasso-Plattner-Institut, University of Potsdam, 2010.

[12] M. Perscheid, D. Cassou, and R. Hirschfeld. Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing*, C5, page accepted. IEEE, 2012.

[13] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. In *Proceedings of the Japan Society for Software Science and Technology Annual Conference*, JSSST, pages 1–17. J-STAGE, 2011.

[14] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. *Journal of the Japan Society for Software Science and Technology on Computer Software*, 29(3):188–211, 2012.

[15] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE, pages 77–86. IEEE, 2010.

[16] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt, and R. Hirschfeld. *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 1st edition, 2008.

[17] C. Queinnec. The Influence of Browsers on Evaluators or, Continuations to Program Web Servers. In *Proceedings of the International Conference on Functional Programming*, ICFP, pages 23–33. ACM, 2000.

[18] G. Rawlinson. *The Significance of Letter Position in Word Recognition*. PhD thesis, University of Nottingham, 1976.

[19] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.

[20] B. Steinert, M. Perscheid, M. Beck, J. Lincke, and R. Hirschfeld. Debugging into Examples: Leveraging Tests for Program Comprehension. In *Proceedings of the International Conference on Testing of Communicating Systems*, TestCom, pages 235–240. Springer, 2009.

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

[22] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2nd edition, 2009.